Contents

OH Stats: Complete Statistical Analysis Guide

1. Introduction

What is oh_stats?

`oh_stats` is a statistical analysis package built on top of `oh_parser`. While `oh_parser` handles the extraction and loading of Occupational Health (OH) profile data from JSON files, `oh_stats` provides the statistical inference layer for analyzing this data.

Why was it built?

The OH profile data has specific characteristics that require specialized handling:

1. **Repeated measures**: Each subject contributes multiple observations across days
2. **Hierarchical structure**: Observations nested within subjects
3. **Multi-modal data**: Multiple sensors (EMG, heart rate, noise, etc.) and questionnaires
4. **Multiple outcomes**: Many metrics measured simultaneously per sensor
5. **Laterality**: Separate measurements for left and right sides (EMG)
6. **Non-normal distributions**: Many outcomes are skewed or bounded

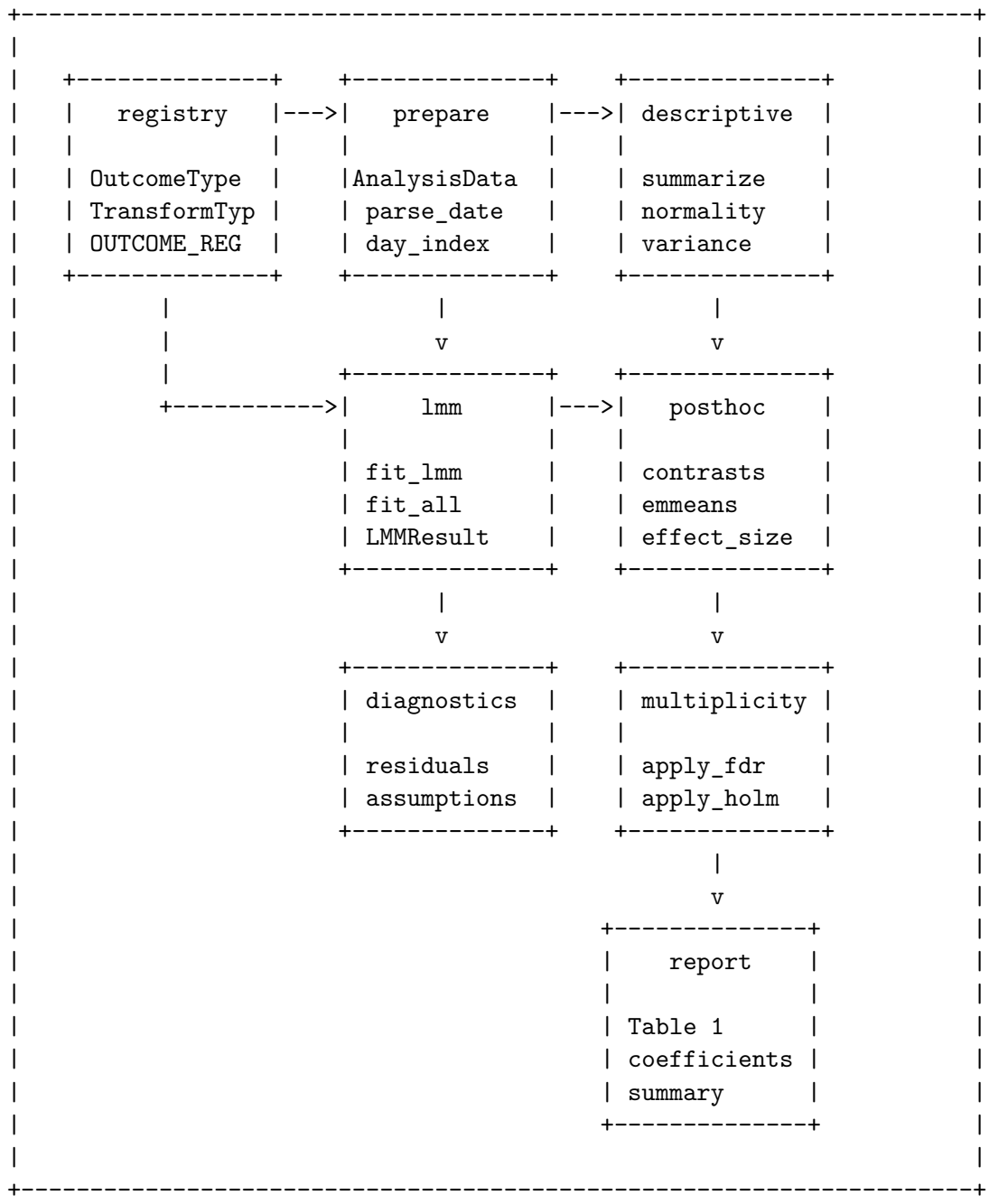
Standard statistical approaches (t-tests, ANOVA) are inappropriate because they: - Ignore the correlation structure within subjects - Inflate Type I error rates - Lose power by ignoring the design

Key Design Principles

1. **Generic data-type based**: Analysis is driven by outcome **data type** (continuous, ordinal, proportion, count), not sensor source. `oh_parser` extracts data from any sensor, `oh_stats` analyzes based on data type.
2. **Registry-driven**: Each outcome is registered with its measurement type, guiding model selection and transform recommendations
3. **Multilevel inference**: Linear Mixed Models (LMM) account for subject-level clustering
4. **Two-layer multiplicity control**: FDR across outcomes, Holm within outcomes
5. **Diagnostic-first**: Check assumptions before trusting results
6. **Function-based architecture**: All data containers use TypedDicts with factory functions for code homogeneity
7. **Discovery-first workflow**: Use discovery functions to understand what data is available before analysis

2. Architecture Overview

```
+-------------------------------------------------------------------+
|                          oh_stats Package                         |
```



Module Dependencies

```
registry.py      <-- No dependencies (foundational)
    |
    v
prepare.py       <-- Uses registry for outcome metadata
    |
    v
descriptive.py   <-- Uses prepare output (AnalysisDataset)
    |
```

```
    v
lmm.py           <-- Uses prepare output + registry for transforms
    |
    v
posthoc.py       <-- Uses LMM results
    |
    v
multiplicity.py  <-- Uses p-values from any source
    |
    v
diagnostics.py   <-- Uses LMM results
    |
    v
report.py        <-- Uses all of the above
```

Function-Based Architecture

The `oh_stats` package uses a **function-based architecture** with TypedDicts for all data containers. This design:

- Maintains code homogeneity with the rest of the `oh_parser` project
- Provides type hints without OOP overhead
- Uses factory functions (`create_*()`) for data container construction
- Uses helper functions instead of instance methods

Example Pattern:

```python
# TypedDict defines the structure
class LMMResult(TypedDict):
    outcome: str
    converged: bool
    # ... other fields

# Factory function creates instances
def create_lmm_result(outcome: str, converged: bool, ...) -> LMMResult:
    return {
        'outcome': outcome,
        'converged': converged,
        # ... other fields
    }

# Helper functions operate on the data
def summarize_lmm_result(result: LMMResult) -> str:
    return f"Model for {result['outcome']}: converged={result['converged']}"
```

3. Data Flow Pipeline

Standard Analysis Workflow

```python
# Step 1: Load OH profiles (oh_parser)
from oh_parser import load_profiles
profiles = load_profiles("/path/to/profiles")

# Step 1b: Discover what data is available (RECOMMENDED)
from oh_stats import get_profile_summary, discover_sensors, discover_questionnaires
print(get_profile_summary(profiles))  # Human-readable overview
sensors = discover_sensors(profiles)  # Dict of sensor -> metrics
quests = discover_questionnaires(profiles)  # Dict of questionnaire domains

# Step 2: Prepare data for analysis (oh_stats)
# Option A: Use convenience wrapper for common cases
from oh_stats import prepare_daily_emg
ds = prepare_daily_emg(profiles, side="both")

# Option B: Use generic function for any sensor
from oh_stats import prepare_sensor_data
ds = prepare_sensor_data(
    profiles,
    sensor="heart_rate",
    base_path="sensor_metrics.heart_rate",
    level_names=["date"],
    value_paths=["HR_BPM_stats.*"]
)

# Step 3: Coverage & missingness report (ALWAYS DO THIS FIRST)
from oh_stats import summarize_outcomes, check_variance, missingness_report
coverage = summarize_outcomes(ds)  # Days per subject, sensors present
variance = check_variance(ds)       # Flag degenerate metrics
missing = missingness_report(ds)    # % missing per outcome

# Step 4: Descriptive statistics
from oh_stats import check_normality
summary = summarize_outcomes(ds, ["EMG_intensity.mean_percent_mvc"])
normality = check_normality(ds, ["EMG_intensity.mean_percent_mvc"])

# Step 5: Fit models
from oh_stats import fit_lmm, fit_all_outcomes
result = fit_lmm(ds, "EMG_intensity.mean_percent_mvc")
all_results = fit_all_outcomes(ds)

# Step 6: Multiple testing correction
from oh_stats import apply_fdr
fdr_results = apply_fdr(all_results)

# Step 7: Post-hoc contrasts (if needed)
from oh_stats import pairwise_contrasts
contrasts = pairwise_contrasts(result, "day_index", ds)

# Step 8: Diagnostics
from oh_stats import residual_diagnostics
diag = residual_diagnostics(result)
```

```python
# Step 9: Generate reports
from oh_stats import descriptive_table, results_summary
table1 = descriptive_table(ds)
summary_df = results_summary(all_results, fdr_results)
```

Critical Pre-Modeling Step: Coverage Report

Before any modeling, always generate a coverage/missingness report. This is the difference between "statistics" and "statistics you can defend in peer review."

```python
# What to check BEFORE modeling:
from oh_stats import summarize_outcomes, check_variance, missingness_report

# 1. Coverage: Days per subject, total observations
print(f"Subjects: {ds['data']['subject_id'].nunique()}")
print(f"Total observations: {len(ds['data'])}")
print(f"Days per subject: {ds['data'].groupby('subject_id')['day_index'].max().describe()}")

# 2. Missingness: % missing per outcome
miss = missingness_report(ds)
high_missing = miss[miss['pct_missing'] > 10]
if len(high_missing) > 0:
    print(f"[WARNING] High missingness (>10%): {high_missing['outcome'].tolist()}")

# 3. Degenerate metrics: Near-zero variance
var = check_variance(ds)
degenerate = var[var['is_degenerate']]['outcome'].tolist()
if degenerate:
    print(f"[EXCLUDE] Exclude from modeling: {degenerate}")
```

The AnalysisDataset Container

The `AnalysisDataset` TypedDict is the central data container:

```python
class AnalysisDataset(TypedDict):
    """Container for analysis-ready data."""
    data: pd.DataFrame          # The actual data
    outcome_vars: List[str]     # Column names of outcomes
    id_var: str                 # Subject identifier column
    time_var: str               # Time variable column
    grouping_vars: List[str]    # Additional grouping (e.g., ["side"])
    sensor: str                 # "emg" or "questionnaire"
    level: str                  # "daily", "session", etc.
    metadata: Dict[str, Any]    # Additional info

# Create using factory function:
ds = create_analysis_dataset(
    data=df,
    outcome_vars=["EMG_intensity.mean_percent_mvc"],
```

```
    id_var="subject_id",
    time_var="day_index",
    grouping_vars=["side"],
    sensor="emg",
    level="daily"
)

# Access using dictionary syntax:
print(f"Subjects: {ds['data']['subject_id'].nunique()}")
print(f"Outcomes: {ds['outcome_vars']}")

# Helper functions for common operations:
description = describe_dataset(ds)  # Get summary
subset = subset_dataset(ds, outcomes=["EMG_intensity.mean_percent_mvc"])
```

Why this design? - Encapsulates all information needed for downstream analysis - Self-documenting: you know what variables are outcomes vs. metadata - Enables consistent handling across all analysis functions - Dictionary access is explicit and homogeneous with rest of project

4. Module Reference

4.1 Registry (registry.py)

Purpose Maps each outcome variable to its statistical properties, enabling automatic model selection and transform recommendations.

Core Types (Enums)

```
class OutcomeType(Enum):
    """Classification of outcome variable measurement type."""
    CONTINUOUS = "continuous"   # Unbounded numeric (e.g., %MVC)
    ORDINAL = "ordinal"         # Ordered categories (e.g., pain 0-10)
    PROPORTION = "proportion"   # Bounded [0, 1] (e.g., rest_percent)
    COUNT = "count"             # Non-negative integers (e.g., gap_count)
    BINARY = "binary"           # 0/1 outcomes
    UNKNOWN = "unknown"         # Not yet classified

class TransformType(Enum):
    """Variance-stabilizing transformations."""
    NONE = "none"               # No transformation
    LOG = "log"                 # log(x), requires x > 0
    LOG1P = "log1p"             # log(1 + x), for x >= 0 with zeros
    SQRT = "sqrt"               # sqrt(x), for x >= 0
    LOGIT = "logit"             # log(x/(1-x)), for proportions (PREFERRED)
    ARCSINE = "arcsine"         # [!] DEPRECATED - see note below

class AnalysisLevel(Enum):
    """Level of data aggregation."""
    DAILY = "daily"
    SESSION = "session"
    SUBJECT = "subject"
```

[!] ARCSINE Transform Deprecation Notice

The arcsine-square-root transform (`arcsin(sqrt(x))`) for proportions is **widely discouraged** in modern statistical practice. It is less interpretable than logit and often unnecessary when you can model the outcome distribution directly (e.g., beta regression).

See: Warton & Hui (2011) "The arcsine is asinine" - *Ecology* 92(1):3-10

Recommendation: Use `LOGIT` for proportions. `ARCSINE` is retained only for legacy compatibility.

The OutcomeInfo TypedDict

```python
class OutcomeInfo(TypedDict, total=False):
    """Metadata for a registered outcome variable."""
    name: str                              # Variable name (required)
    outcome_type: OutcomeType              # Measurement type (required)
    transform: TransformType               # Recommended transform
    level: str                             # Aggregation level
    is_primary: bool                       # Primary endpoint?
    description: str                       # Human-readable description
    min_value: Optional[float]             # Theoretical minimum
    max_value: Optional[float]             # Theoretical maximum
    unit: str                              # Measurement unit

# Create using factory function:
info = create_outcome_info(
    name="EMG_intensity.mean_percent_mvc",
    outcome_type=OutcomeType.CONTINUOUS,
    transform=TransformType.NONE,
    level="daily",
    is_primary=True,
    description="Mean muscle activation as percentage of MVC",
    unit="%MVC"
)
```

Pre-registered Outcomes The following EMG outcomes are pre-registered:

Outcome	Type	Transform	Notes
mean_percent_mvc	CONTINUOUS	NONE	Primary
max_percent_mvc	CONTINUOUS	NONE	
iemg_percent_seconds	CONTINUOUS	LOG	Right-skewed
apdf.active.p50	CONTINUOUS	NONE	Primary
rest_percent	PROPORTION	LOGIT	Bounded [0,1]
gap_count	COUNT	LOG1P	Fallback*

Pre-registered Questionnaire Outcomes (v0.3.0+):

Outcome	Type	Transform	Notes
copsoq_quant_demands	CONTINUOUS	NONE	Primary

Outcome	Type	Transform	Notes
copsoq_work_pace	CONTINUOUS	NONE	
copsoq_emotional	CONTINUOUS	NONE	
copsoq_influence	CONTINUOUS	NONE	
copsoq_meaning	CONTINUOUS	NONE	
copsoq_social_support	CONTINUOUS	NONE	Primary
mueq_work_station	CONTINUOUS	NONE	
mueq_body_posture	CONTINUOUS	NONE	Primary
mueq_job_control	CONTINUOUS	NONE	
mueq_job_demands	CONTINUOUS	NONE	
mueq_breaks	CONTINUOUS	NONE	
mueq_total	CONTINUOUS	NONE	Primary
rosa_total	ORDINAL	NONE	Primary
ipaq_met_min_week	CONTINUOUS	LOG1P	Right-skewed
ipaq_category	ORDINAL	NONE	
ospaq_sitting_pct	PROPORTION	LOGIT	Bounded [0,1]
ospaq_standing_pct	PROPORTION	LOGIT	Bounded [0,1]
ospaq_walking_pct	PROPORTION	LOGIT	Bounded [0,1]
nprs_pain_current	ORDINAL	NONE	Primary
nprs_pain_average	ORDINAL	NONE	
daily_workload_*	ORDINAL	NONE	
daily_environ_*	CONTINUOUS	NONE	

COUNT outcomes: The `LOG1P` transform + Gaussian LMM is a **pragmatic fallback, not the principled solution. True count data should use Poisson/Negative Binomial GLMMs. This is acceptable for exploratory analysis but consider proper count models for publication.

Primary vs. Secondary Outcomes: Multiplicity Strategy The `is_primary` field in the registry enables **differentiated multiplicity control**:

Outcome Class	Correction Method	Rationale
Primary (confirmatory)	FWER (Holm-Bonferroni)	Strong control for main hypotheses
Secondary (exploratory)	FDR (Benjamini-Hochberg)	Discovery-oriented, accepts some false positives

Enforcement Pattern:

```
from oh_stats import list_outcomes, apply_fdr, apply_holm

# Get outcomes by class
primary = list_outcomes(is_primary=True)
secondary = list_outcomes(is_primary=False)
```

```
# Fit models
primary_results = fit_all_outcomes(ds, outcomes=primary)
secondary_results = fit_all_outcomes(ds, outcomes=secondary)

# Apply appropriate correction
primary_corrected = apply_holm(primary_results)      # FWER for primaries
secondary_corrected = apply_fdr(secondary_results)   # FDR for secondaries
```

Default Report Behavior:

- `results_summary()` displays primary outcomes first
- Primary outcomes are flagged in output tables
- This matches how occupational health papers are typically structured and defended

Recommendation: Pre-specify 2-4 primary outcomes based on research questions. Everything else is exploratory. This distinction should be documented in your analysis plan BEFORE seeing results.

Usage

```
from oh_stats import get_outcome_info, register_outcome, OutcomeType, TransformType

# Get info for existing outcome
info = get_outcome_info("EMG_intensity.mean_percent_mvc")
print(f"Type: {info['outcome_type']}")  # CONTINUOUS
print(f"Transform: {info['transform']}")  # NONE

# Register a new outcome
register_outcome(
    name="my_custom_metric",
    outcome_type=OutcomeType.CONTINUOUS,
    transform=TransformType.LOG,
    level="daily",
    description="Custom metric from EMG processing"
)

# List all outcomes
from oh_stats import list_outcomes
outcomes = list_outcomes(outcome_type=OutcomeType.CONTINUOUS)

# List questionnaire-specific outcomes (v0.3.0+)
from oh_stats import get_questionnaire_outcomes, get_daily_outcomes, get_single_instance_outcomes

# All questionnaire outcomes (COPSOQ, MUEQ, ROSA, IPAQ, OSPAQ, NPRS, etc.)
questionnaire_outcomes = get_questionnaire_outcomes()

# Daily repeated measures (workload, pain, environmental)
daily_outcomes = get_daily_outcomes()

# Single-instance baseline measures (COPSOQ, MUEQ, ROSA, IPAQ, OSPAQ)
baseline_outcomes = get_single_instance_outcomes()
```

Why a Registry?

1. **Automatic model family dispatch**: Maps outcome types to appropriate models (see table below)
2. **Transform recommendations**: Skewed data -> LOG transform
3. **Validation**: Catch misspecified analyses early
4. **Documentation**: Centralized knowledge about each variable

Model Family Dispatch Table

Outcome	Preferred	Current	Notes
CONTINUOUS	Gaussian LMM	OK	statsmodels
ORDINAL	Ordered logit	STUB	Future
PROPORTION	Beta/Logit+LMM	OK	Not logistic
COUNT	Poisson/NB GLMM	STUB	LOG1P fallback
BINARY	Logistic GLMM	STUB	True logistic

[!] Important Distinction: "Logistic regression" is for **binary** (0/1) outcomes only. For **proportions** (values in 0-1), we use the **logit transform** on the continuous outcome, then fit a **Gaussian** LMM on the transformed scale. This is statistically defensible and often works well in practice.

4.2 Data Preparation (prepare.py)

Purpose Transform raw `oh_parser` output into analysis-ready `AnalysisDataset` dictionaries.

Design Philosophy: Generic Data-Type Based Analysis The `oh_stats` package uses a **generic, data-type based** approach:

- **`oh_parser`** extracts data from any sensor (EMG, heart rate, noise, accelerometer, etc.)
- **`oh_stats`** analyzes based on **data type** (continuous, ordinal, proportion, count)

This means you don't need sensor-specific functions for each sensor. Instead: 1. Use **discovery functions** to see what data is available 2. Use **`prepare_sensor_data()`** (generic) or convenience wrappers for extraction 3. Use **`register_outcome()`** to specify the data type for statistical analysis

Discovery Functions (Start Here!) Before preparing data, discover what's available in your profiles:

`discover_sensors()`

```
def discover_sensors(profiles: Dict[str, dict]) -> Dict[str, List[str]]:
    """
    Discover available sensors and their metric keys from OH profiles.

    Parameters
    ----------
    profiles : dict
        Output from oh_parser.load_profiles()

    Returns
    -------
    dict
        Mapping sensor names to lists of available metric keys

    Example
    -------
    >>> sensors = discover_sensors(profiles)
    >>> print(sensors)
    {'heart_rate': ['HR_BPM_stats', 'HR_ratio_stats', ...],
     'noise': ['Noise_statistics', ...],
     'emg': ['EMG_intensity', 'EMG_apdf', ...]}
    """
```

`discover_questionnaires()`

```
def discover_questionnaires(profiles: Dict[str, dict]) -> Dict[str, Dict[str, List[str]]]:
    """
    Discover available questionnaire domains and their fields.

    Parameters
    ----------
    profiles : dict
        Output from oh_parser.load_profiles()
```

```
    Returns
    -------
    dict
        With 'single_instance' and 'daily' keys, each mapping
        domain names to lists of field names

    Example
    -------
    >>> quests = discover_questionnaires(profiles)
    >>> print(quests['single_instance'].keys())
    dict_keys(['personal', 'biomechanical', 'psychosocial', 'environmental'])
    >>> print(quests['daily'].keys())
    dict_keys(['workload', 'pain'])
    """
```

get_profile_summary()

```
def get_profile_summary(profiles: Dict[str, dict]) -> str:
    """
    Generate a human-readable summary of available data.

    Example
    -------
    >>> print(get_profile_summary(profiles))
    OH Profile Summary (42 subjects)
    ================================================

    SENSOR DATA:
      emg: 15 metrics
      heart_rate: 8 metrics
      noise: 6 metrics

    SINGLE-INSTANCE QUESTIONNAIRES:
      personal: 31 fields
      biomechanical: 73 fields
      psychosocial: 4 fields

    DAILY QUESTIONNAIRES:
      workload: 6 fields
      pain: 12 fields
    """
```

Generic Sensor Data Preparation

prepare_sensor_data()

```
def prepare_sensor_data(
    profiles: Dict[str, dict],
    sensor: str,                          # Sensor name (e.g., "heart_rate")
    base_path: str,                       # JSON path (e.g., "sensor_metrics.heart_rate")
```

```
    level_names: List[str],              # Nested level names (e.g., ["date"])
    value_paths: List[str],              # Glob patterns for values
    level_filter: Optional[Dict[str, str]] = None,  # Filter by level value
    side: str = "both",                  # Side handling
    add_day_index: bool = True,
    add_weekday: bool = True,
) -> AnalysisDataset:
    """
    Generic preparation function for ANY sensor data.

    This is the core function - modality-specific functions like
    prepare_daily_emg() are convenience wrappers around this.

    Example: Prepare heart rate data
    --------------------------------
    >>> ds = prepare_sensor_data(
    ...     profiles,
    ...     sensor="heart_rate",
    ...     base_path="sensor_metrics.heart_rate",
    ...     level_names=["date"],
    ...     value_paths=["HR_BPM_stats.*", "HR_ratio_stats.*"]
    ... )

    Example: Prepare noise data
    ---------------------------
    >>> ds = prepare_sensor_data(
    ...     profiles,
    ...     sensor="noise",
    ...     base_path="sensor_metrics.noise",
    ...     level_names=["date"],
    ...     value_paths=["Noise_statistics.*"]
    ... )

    Example: Prepare environment data
    ---------------------------------
    >>> ds = prepare_sensor_data(
    ...     profiles,
    ...     sensor="environment",
    ...     base_path="sensor_metrics.environment",
    ...     level_names=["date"],
    ...     value_paths=["*"]
    ... )
    """
```

EMG Convenience Wrappers For EMG data (the most common use case), convenience wrappers are provided:

`prepare_daily_emg()`

```
def prepare_daily_emg(
    profiles: Dict[str, dict],
    side: str = "both",           # "left", "right", "both", "average"
```

```
    add_day_index: bool = True,    # Add ordinal day within subject
    add_weekday: bool = True,      # Add day name
) -> AnalysisDataset:
    """
    Prepare EMG data at the daily aggregation level.

    Parameters
    ----------
    profiles : dict
        Output from oh_parser.load_profiles()
    side : str
        How to handle laterality:
        - "left": Only left side data
        - "right": Only right side data
        - "both": Keep as separate rows (doubles observations)
        - "average": Average left and right into single value
    add_day_index : bool
        Add within-subject ordinal day (1, 2, 3, ...)
    add_weekday : bool
        Add day-of-week name column

    Returns
    -------
    AnalysisDataset
        Ready for analysis with all EMG outcomes (TypedDict)
    """
```

Side Handling Strategies:

Strategy	Effect	When to Use
`"both"`	Left and right as separate rows	When laterality is of interest
`"average"`	Mean of left/right	When laterality is nuisance
`"left"` / `"right"`	Keep only one side	When sides have different meaning

> **[!] Statistical Implication of `side="both"`**
>
> When keeping both sides as separate rows, left and right observations from the **same subject-day are NOT independent**. The current model with only a subject-level random intercept doesn't fully account for this within-day correlation.
>
> **Options to handle this properly:** 1. **Add subject-day clustering** (more complex model): `(1|subject_id/day_index)` 2. **Use `side="average"`** if laterality is not the research question 3. **Analyze each side separately** as a sensitivity check
>
> For the current EMG analysis where `side` is a fixed effect of interest, the subject random intercept is a reasonable approximation.

Multi-Channel Correlation: The Subject-Day Clustering Pattern The side handling issue is a specific case of a **general pattern**: any sensor with multiple channels per day (left/right EMG, multi-axis accelerometer, multiple device placements) creates correlated observations that share a subject-day context.

Documented Variance Structure Options:

Model Complexity	Formula	Use When
Simple (current)	`(1|subject_id)`	Single channel per day, or channels averaged
Intermediate	`(1|subject_id) + (1|subject_day_id)`	Multiple channels, day-level correlation matters
Full nested	`(1|subject_id/day_index/channel)`	Complex multi-level structure

Implementation Pattern:

```
# Create subject-day identifier for intermediate clustering
ds['data']['subject_day_id'] = (
    ds['data']['subject_id'].astype(str) + '_' +
    ds['data']['day_index'].astype(str)
)

# Fit with subject-day random effect (future enhancement)
result = fit_lmm(
    ds, outcome,
    random_intercept="subject_id",
    nested_random="subject_day_id",  # Planned feature
)
```

> **Current Status**: v0.2.0 supports subject-level random intercept only. Subject-day clustering is documented here for (1) methodological transparency and (2) preventing misuse when multi-channel data is involved. Full nested random effects are planned for v0.3.0.

Example:

```
# Keep both sides (320 obs = 160 days x 2 sides)
ds_both = prepare_daily_emg(profiles, side="both")
print(ds_both['data'].shape)  # (320, 27)

# Average sides (160 obs = 160 days)
ds_avg = prepare_daily_emg(profiles, side="average")
print(ds_avg['data'].shape)  # (160, 25)
```

`create_analysis_dataset()` Factory function for creating AnalysisDataset containers:

```
def create_analysis_dataset(
    data: pd.DataFrame,
    outcome_vars: List[str],
    id_var: str = "subject_id",
```

```
    time_var: str = "day_index",
    grouping_vars: Optional[List[str]] = None,
    sensor: str = "emg",
    level: str = "daily",
    metadata: Optional[Dict[str, Any]] = None
) -> AnalysisDataset:
    """Create an AnalysisDataset TypedDict."""
```

Helper Functions `describe_dataset()` - Get summary statistics for an AnalysisDataset.

```
def describe_dataset(ds: AnalysisDataset) -> Dict[str, Any]:
    """Get summary statistics for an AnalysisDataset."""
```

`subset_dataset()` - Subset an AnalysisDataset by outcomes, subjects, or time range.

```
def subset_dataset(
    ds: AnalysisDataset,
    outcomes: Optional[List[str]] = None,
    subjects: Optional[List[str]] = None,
    time_range: Optional[Tuple[int, int]] = None
) -> AnalysisDataset:
```

`validate_dataset()` - Validate an AnalysisDataset structure. Returns list of warnings.

```
def validate_dataset(ds: AnalysisDataset) -> List[str]:
```

Day Index Computation Within each subject, days are assigned ordinal indices:

```
Subject 103:
  2025-10-13 -> day_index = 1
  2025-10-14 -> day_index = 2
  2025-10-15 -> day_index = 3
  ...

Subject 104:
  2025-09-22 -> day_index = 1  (different calendar date!)
  2025-09-23 -> day_index = 2
  ...
```

Why ordinal, not calendar day? - Subjects start on different dates - Interest is in "day 1 vs day 5 of monitoring," not "October 13 vs October 17" - Makes between-subject comparison meaningful

4.3 Descriptive Statistics (descriptive.py)

Purpose Summarize data and check assumptions before modeling.

Data Containers (TypedDicts)

```python
class NormalityResult(TypedDict):
    """Result of normality testing for an outcome."""
    outcome: str
    n: int
    n_missing: int
    skewness: float
    kurtosis: float
    shapiro_stat: Optional[float]
    shapiro_p: Optional[float]
    is_normal: bool
    recommended_transform: str

class VarianceCheckResult(TypedDict):
    """Result of variance/degeneracy check for an outcome."""
    outcome: str
    n: int
    n_unique: int
    pct_mode: float
    variance: float
    is_degenerate: bool
    reason: str
```

Main Functions

summarize_outcomes()

```python
def summarize_outcomes(
    ds: AnalysisDataset,
    outcomes: Optional[List[str]] = None,  # None = all outcomes
    by_group: bool = False,                 # Group by side?
) -> pd.DataFrame:
    """
    Generate comprehensive summary statistics.

    Returns
    -------
    DataFrame with columns:
        outcome, n, n_missing, pct_missing, mean, std, min,
        p25, median, p75, max, skewness, kurtosis, cv
    """
```

Output Example:

outcome	n	n_missing	mean	std	min	p25	median
EMG_intensity.mean_percent_mvc	320	0	9.126	6.992	1.080	4.424	7.443
EMG_apdf.active.p50	320	0	6.811	5.901	0.696	3.015	5.378

`check_normality()`

```
def check_normality(
    ds: AnalysisDataset,
    outcomes: Optional[List[str]] = None,
    alpha: float = 0.05,
) -> pd.DataFrame:
    """
    Test normality of each outcome using Shapiro-Wilk test.

    Returns
    -------
    DataFrame with:
        outcome, n, skewness, shapiro_stat, shapiro_p,
        is_normal, recommended_transform
    """
```

[!] Don't Over-Interpret Normality Tests

The Shapiro-Wilk p-value should be treated as a **signal, not a verdict**:

- **With moderate-to-large N**: Tests become "significant" for trivial deviations
- **With small N**: Tests lack power to detect real departures

Better approach: 1. Use **visual diagnostics** first (QQ plots, histograms) 2. Treat Shapiro p-value as one input, not a gatekeeper 3. Check whether fixed-effect estimates are **stable** under transformation 4. LMMs are fairly robust to mild normality violations

`check_variance()`

```
def check_variance(
    ds: AnalysisDataset,
    outcomes: Optional[List[str]] = None,
    threshold_unique: int = 5,        # Minimum distinct values
    threshold_mode_pct: float = 95.0, # Maximum mode percentage
) -> pd.DataFrame:
    """
    Detect degenerate or near-constant outcomes.

    A variable is DEGENERATE if:
    - Fewer than 5 unique values, OR
    - Mode accounts for >95% of observations

    Degenerate outcomes should NOT be modeled.
    """
```

4.4 Linear Mixed Models (lmm.py)

Purpose Fit Linear Mixed Effects Models accounting for repeated measures within subjects.

Why Linear Mixed Models? The Problem: Each subject provides multiple observations (days). These observations are **not independent** - values from the same subject are more similar than values from different subjects.

Standard approaches fail: - **T-tests/ANOVA**: Assume independence -> inflated Type I error - **Paired t-tests**: Only handle 2 time points - **Repeated measures ANOVA**: Requires complete data, sphericity assumption

LMM Solution:

```
Y_ij = B0 + B1*Day_ij + B2*Side_ij + u_i + e_ij

Where:
- Y_ij: Outcome for subject i on observation j
- B0: Grand mean (intercept)
- B1: Effect of day
- B2: Effect of side
- u_i ~ N(0, sigma^2_u): Subject-specific random intercept
- e_ij ~ N(0, sigma^2): Residual error
```

The LMMResult TypedDict

```python
class LMMResult(TypedDict):
    """Container for all model output."""
    outcome: str                    # Which outcome was modeled
    model: Any                      # The statsmodels model object
    formula: str                    # Formula used (e.g., "Y ~ C(day)")
    coefficients: pd.DataFrame      # Fixed effects estimates (Wald tests)
    fit_stats: Dict[str, float]     # AIC, BIC, log-likelihood, LRT results
    random_effects: Dict[str, float]  # Variance components + ICC
    n_obs: int                      # Number of observations
    n_groups: int                   # Number of subjects
    converged: bool                 # Did optimization succeed?
    transform_applied: str          # What transform was used
    warnings: List[str]             # Any issues encountered

# Create using factory function:
result = create_lmm_result(
    outcome="EMG_intensity.mean_percent_mvc",
    model=fitted_model,
    formula="Y ~ C(day_index) + C(side)",
    coefficients=coef_df,
    fit_stats={
        'aic': 478.4,
        'bic': 502.1,
        'loglik': -234.2,
        'lrt_stat': 12.5,        # LRT chi-square for day effect
        'lrt_df': 4,             # Degrees of freedom
```

```
        'lrt_pvalue': 0.014,      # P-value for FDR correction
    },
    random_effects={
        'group_var': 24.0,        # sigma^2_u
        'residual_var': 23.9,     # sigma^2
        'icc': 0.50,              # Intraclass correlation
    },
    n_obs=320,
    n_groups=37,
    converged=True,
    transform_applied="none",
    warnings=[]
)

# Access using dictionary syntax:
print(f"AIC: {result['fit_stats']['aic']}")
print(f"ICC: {result['random_effects']['icc']}")

# Summarize result:
print(summarize_lmm_result(result))
```

Main Functions

`fit_lmm()`

```
def fit_lmm(
    ds: AnalysisDataset,
    outcome: str,
    fixed_effects: Optional[List[str]] = None,
    random_intercept: str = "subject_id",
    transform: Optional[TransformType] = None,
    day_as_categorical: bool = True,
    include_side: bool = True,
    formula: Optional[str] = None,
    reml: bool = False,
) -> LMMResult:
    """
    Fit a Linear Mixed Model for a single outcome.

    Parameters
    ----------
    ds : AnalysisDataset
        Prepared data
    outcome : str
        Name of outcome variable to model
    fixed_effects : list, optional
        Custom fixed effects. Default: [C(day_index), C(side)]
    random_intercept : str
        Grouping variable for random intercept (default: subject_id)
    transform : TransformType, optional
        Override registry transform recommendation
    day_as_categorical : bool
```

```
        Treat day as categories (True) or linear trend (False)
    include_side : bool
        Include side effect if present
    formula : str, optional
        Complete formula override (advanced)
    reml : bool
        Use REML estimation. Default False (ML for valid AIC/BIC)

    Returns
    -------
    LMMResult
        TypedDict containing model output
    """
```

prepare_baseline_questionnaires() (v0.3.0+)

```
def prepare_baseline_questionnaires(
    profiles: Dict[str, dict],
    questionnaire_type: str = "all",   # "copsoq", "mueq", "rosa", "ipaq", "ospaq", or "all"
    compute_composites: bool = True,    # Auto-compute dimension scores
) -> AnalysisDataset:
    """
    Prepare single-instance baseline questionnaire data.

    These questionnaires are administered once per subject:
    - COPSOQ: Copenhagen Psychosocial Questionnaire (work stress)
    - MUEQ: Musculoskeletal-Utrecht Ergonomic Questionnaire
    - ROSA: Rapid Office Strain Assessment (1-10)
    - IPAQ: International Physical Activity Questionnaire
    - OSPAQ: Occupational Sitting and Physical Activity Questionnaire
    """
```

prepare_daily_workload() (v0.3.0+)

```
def prepare_daily_workload(
    profiles: Dict[str, dict],
    aggregate_to_composite: bool = True,  # Create composite workload score
) -> AnalysisDataset:
    """
    Prepare daily workload Likert items (repeated measures).

    Items are rated 1-5 daily, covering:
    - Physical demands
    - Mental demands
    - Time pressure
    - Work interruptions
    """
```

prepare_daily_pain() (v0.3.0+)

```python
def prepare_daily_pain(
    profiles: Dict[str, dict],
    body_regions: Optional[List[str]] = None,  # Filter specific regions
) -> AnalysisDataset:
    """
    Prepare daily NPRS pain ratings (0-10 scale).

    Tracks pain by body region over time.
    """
```

compute_composite_score() (v0.3.0+)

```python
def compute_composite_score(
    df: pd.DataFrame,
    items: List[str],
    reverse_items: Optional[List[str]] = None,
    scale_max: int = 5,
    missing_rule: str = "half",  # "half", "any", "none"
    output_scale: Tuple[int, int] = (0, 100),
) -> pd.Series:
    """
    Compute composite scores from Likert items.

    Features:
    - Automatic reverse coding for negatively-worded items
    - Configurable missing data rules
    - Rescaling to 0-100 standard
    """
```

align_sensor_questionnaire() (v0.3.0+)

```python
def align_sensor_questionnaire(
    sensor_ds: AnalysisDataset,
    questionnaire_ds: AnalysisDataset,
    join_on: List[str] = ["subject_id", "date"],
) -> AnalysisDataset:
    """
    Merge sensor data (EMG) with questionnaire data.

    Enables multi-modal analysis: "Does perceived workload predict EMG?"
    """
```

Day as Categorical vs. Linear:

```python
# Categorical (default): Estimate separate effect for each day
# Formula: Y ~ C(day_index) + C(side)
# Coefficients: Day2 vs Day1, Day3 vs Day1, Day4 vs Day1, ...
# Use when: Pattern over days is non-linear

# Linear trend: Assume linear change per day
```

```
# Formula: Y ~ day_index + C(side)
# Coefficients: slope (change per day)
# Use when: Expecting monotonic trend

result_cat = fit_lmm(ds, outcome, day_as_categorical=True)
result_lin = fit_lmm(ds, outcome, day_as_categorical=False)
```

fit_all_outcomes()

```
def fit_all_outcomes(
    ds: AnalysisDataset,
    outcomes: Optional[List[str]] = None,
    outcome_type: Optional[OutcomeType] = None,
    skip_degenerate: bool = True,
    max_outcomes: Optional[int] = None,
    **kwargs,
) -> Dict[str, LMMResult]:
    """
    Fit LMM for multiple outcomes in batch.

    Returns
    -------
    dict
        {outcome_name: LMMResult} for each fitted model
    """
```

Understanding the Output Coefficients DataFrame:

```
                term  estimate  std_error   z_value   p_value  ci_lower  ci_upper
C(day_index)[T.2] -0.411170   0.825203 -0.498265  0.618297 -2.028538  1.206198
C(day_index)[T.3] -0.028058   0.838860 -0.033448  0.973318 -1.672194  1.616078
C(day_index)[T.4] -1.930854   0.840079 -2.298419  0.021538 -3.577380 -0.284329
 C(side)[T.right]  0.902430   0.549862  1.641193  0.100757 -0.175280  1.980140
        Intercept  9.405613   1.035062  9.087006  0.000000  7.376929 11.434297
```

Interpretation: - `Intercept (9.41)`: Mean %MVC on Day 1, Left side - `C(day_index)[T.4] (-1.93, p=0.02)`: Day 4 is 1.93 %MVC lower than Day 1 (significant) - `C(side)[T.right] (0.90, p=0.10)`: Right side is 0.90 %MVC higher (not significant)

Random Effects (accessed via dictionary):

```python
print(result['random_effects'])
# {
#     'group_var': 24.048,      # Between-subject variance (sigma^2_u)
#     'residual_var': 23.884,   # Within-subject variance (sigma^2)
#     'icc': 0.502              # Intraclass correlation
# }
```

ICC (Intraclass Correlation):

```
ICC = sigma^2_u / (sigma^2_u + sigma^2)
    = 24.048 / (24.048 + 23.884)
    = 0.502
```

Interpretation:

- ICC = 0.50 means **50% of total variance is between subjects**
- This justifies using mixed models - observations within subjects are indeed correlated

4.5 Post-Hoc Comparisons (posthoc.py)

Purpose When an overall effect is significant, identify which specific comparisons drive the effect.

The ContrastResult TypedDict

```
class ContrastResult(TypedDict):
    """Result of a single contrast comparison."""
    contrast: str               # e.g., "Day1-Day4"
    estimate: float             # Difference estimate
    std_error: float            # Standard error
    z_value: float              # Test statistic
    p_value: float              # Raw p-value
    p_adjusted: float           # Adjusted p-value
    ci_lower: float             # Lower CI bound
    ci_upper: float             # Upper CI bound
    cohens_d: Optional[float]   # Effect size

# Factory function:
contrast = create_contrast_result(
    contrast="Day1-Day4",
    estimate=1.931,
    std_error=0.840,
    z_value=2.298,
    p_value=0.022,
    p_adjusted=0.043,
    ci_lower=0.284,
    ci_upper=3.577,
    cohens_d=0.276
)

# Summarize:
print(summarize_contrast_result(contrast))
```

Main Functions

`pairwise_contrasts()`

```
def pairwise_contrasts(
    result: LMMResult,
    factor: str,                    # "day_index" or "side"
    ds: AnalysisDataset,
    adjustment: str = "holm",       # "holm", "bonferroni", "none"
) -> pd.DataFrame:
    """
    Compute all pairwise comparisons for a factor.

    Returns DataFrame with columns:
        contrast, estimate, std_error, z_value, p_value,
        p_adjusted, ci_lower, ci_upper, cohens_d
    """
```

compute_emmeans()

```python
def compute_emmeans(
    result: LMMResult,
    factor: str,
    ds: AnalysisDataset,
) -> pd.DataFrame:
    """
    Compute Estimated Marginal Means (EMMs).

    EMMs are model-predicted means for each level of a factor,
    averaging over other factors.
    """
```

compute_effect_size()

```python
def compute_effect_size(
    contrast_estimate: float,
    pooled_sd: float,
    effect_type: str = "cohens_d",
) -> float:
    """
    Compute standardized effect size.

    Cohen's d interpretation:
        |d| < 0.2: Negligible
        0.2 <= |d| < 0.5: Small
        0.5 <= |d| < 0.8: Medium
        |d| >= 0.8: Large
    """
```

Effect Size Definition for Mixed Models In LMMs, variance is decomposed into components (subject random intercept + residual), so there is no single "pooled SD." We adopt the following **explicit definition**:

Default: Residual-Standardized Effect Size

`d = Delta / sigma_residual`

Where: - `Delta` = contrast estimate (e.g., Day1 - Day4 difference) - `sigma_residual` = square root of residual variance from LMM

This standardizes by **within-subject variability**, which is appropriate for repeated-measures designs where subject-level variance is nuisance.

Alternative Definitions (for sensitivity analysis):

Definition	Formula	When to Use
Residual-standardized (default)	d = Delta / sigma_res	Within-subject comparisons

Definition	Formula	When to Use
Total-standardized	d = Delta / sqrt(sigma^2_u + sigma^2)	Between-subject interpretability
Raw units	Report Delta + 95% CI directly	Clinical/practical interpretation

Recommendation: Always report raw-unit effects with confidence intervals as the primary result. Cohen's d is supplementary for readers who want standardized comparisons across studies.

4.6 Multiplicity Correction (multiplicity.py)

Purpose Control false positive rate when testing many hypotheses simultaneously.

Critical: Which P-Value Feeds FDR? LMM output contains **multiple p-values** (one per coefficient). For outcome-wise FDR correction, we need **one p-value per outcome** representing the overall "day effect."

Default Strategy (Omnibus Test):

We use a **Likelihood Ratio Test (LRT)** comparing: - **Full model**: `Y ~ C(day_index) + C(side) + (1|subject_id)` - **Reduced model**: `Y ~ C(side) + (1|subject_id)` (day removed)

The LRT p-value tests: "Does including day improve model fit?" This is the p-value used for across-outcome FDR correction.

```
# LRT is computed automatically in fit_lmm() and stored in:
result['fit_stats']['lrt_pvalue']  # P-value for day effect (omnibus)
result['fit_stats']['lrt_stat']    # Chi-square statistic
result['fit_stats']['lrt_df']      # Degrees of freedom
```

> [!] **Important Distinction**
>
> - **LRT p-value (omnibus)**: Used for FDR across outcomes ("Is there ANY day effect?")
> - **Coefficient p-values (Wald)**: Used for interpretation tables ("Which specific days differ?")
>
> Never apply FDR correction to arbitrary coefficient-level p-values - this conflates the multiplicity layers and inflates false discovery.

Two-Layer Correction Strategy

```
Layer 1: ACROSS outcomes (FDR correction)
+-- Outcome 1: p_raw = 0.001 -> p_adj = 0.005 [PASS]
+-- Outcome 2: p_raw = 0.02  -> p_adj = 0.04  [PASS]
+-- Outcome 3: p_raw = 0.08  -> p_adj = 0.10  [FAIL]
+-- ...

Layer 2: WITHIN significant outcomes (Holm correction for post-hoc)
+-- Outcome 1 post-hocs:
|   +-- Day1-Day2: p = 0.50 -> p_adj = 0.50
|   +-- Day1-Day4: p = 0.02 -> p_adj = 0.04 [PASS]
|   +-- ...
+-- Outcome 2 post-hocs:
    +-- ...
```

Main Functions

`apply_fdr()`

```
def apply_fdr(
    results: Dict[str, LMMResult],
    alpha: float = 0.05,
    method: str = "fdr_bh",       # Benjamini-Hochberg
) -> pd.DataFrame:
    """
    Apply False Discovery Rate correction across outcomes.

    Returns DataFrame with:
        outcome, p_raw, p_adjusted, significant
    """
```

apply_holm()

```
def apply_holm(
    p_values: List[float],
    alpha: float = 0.05,
) -> pd.DataFrame:
    """
    Apply Holm-Bonferroni correction (step-down procedure).
    More powerful than Bonferroni while controlling FWER.
    """
```

adjust_pvalues()

```
def adjust_pvalues(
    p_values: np.ndarray,
    method: str = "fdr_bh",
) -> np.ndarray:
    """
    Wrapper for multiple testing correction methods.

    Supported methods:
    - "fdr_bh": Benjamini-Hochberg FDR
    - "fdr_by": Benjamini-Yekutieli FDR (more conservative)
    - "holm": Holm-Bonferroni FWER
    - "bonferroni": Classical Bonferroni FWER
    - "none": No adjustment
    """
```

4.7 Model Diagnostics (diagnostics.py)

Purpose Verify that model assumptions are met before trusting results.

The DiagnosticsResult TypedDict

```
class DiagnosticsResult(TypedDict):
    """Comprehensive diagnostics output for an LMM."""
    outcome: str
    residuals: np.ndarray           # Raw residuals
    fitted: np.ndarray              # Fitted values
    standardized: np.ndarray        # Standardized residuals
    normality_stat: float           # Shapiro-Wilk statistic
    normality_p: float              # Normality p-value
    heteroscedasticity_stat: Optional[float]  # Breusch-Pagan statistic
    heteroscedasticity_p: Optional[float]     # Heteroscedasticity p-value
    outlier_indices: List[int]      # Indices of outliers (|z| > 3)
    n_outliers: int
    assumptions_met: bool           # Overall assessment

# Factory function:
diag = create_diagnostics_result(
    outcome="EMG_intensity.mean_percent_mvc",
    residuals=residuals,
    fitted=fitted_values,
    standardized=std_residuals,
    normality_stat=0.98,
    normality_p=0.12,
    heteroscedasticity_stat=1.45,
    heteroscedasticity_p=0.23,
    outlier_indices=[45, 156],
    n_outliers=2,
    assumptions_met=True
)

# Summarize:
print(summarize_diagnostics(diag))
```

Main Functions

`residual_diagnostics()`

```
def residual_diagnostics(result: LMMResult) -> DiagnosticsResult:
    """
    Comprehensive residual analysis.

    Returns DiagnosticsResult TypedDict with:
        - residuals: Raw residuals
        - fitted: Fitted values
        - standardized: Standardized residuals
        - normality_stat/p: Shapiro-Wilk test (use as signal, not verdict)
        - heteroscedasticity_stat/p: Breusch-Pagan test (screening only)
```

```
        - outlier_indices: Observations with |z| > 3
    """
```

[!] Heteroscedasticity Testing Caveat

The Breusch-Pagan test assumes **independent residuals**, but LMM residuals have correlation structure (observations within subjects are dependent). Therefore:

- Use BP as a **screening heuristic**, not a definitive test
- The **residual vs. fitted plot** is more informative for detecting patterns
- If heteroscedasticity is suspected:
  1. Try variance-stabilizing transforms (LOG, SQRT)
  2. Consider robust standard errors as sensitivity analysis
  3. Check if patterns differ by subject or time

The same "signal, not verdict" philosophy from normality testing applies here.

`check_assumptions()`

```
def check_assumptions(
    result: LMMResult,
    alpha: float = 0.05,
) -> Dict[str, Dict[str, Any]]:
    """
    Automated assumption checking with pass/fail status.

    Note: "FAIL" means the test flagged an issue, not that
    the model is invalid. Use judgment and visual inspection.
    """
```

Example (recommended approach):

```
diag = residual_diagnostics(result)

# 1. Check summary
print(summarize_diagnostics(diag))

# 2. Visual check (most important)
import matplotlib.pyplot as plt
from scipy import stats

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

# QQ plot
stats.probplot(diag['standardized'], dist="norm", plot=axes[0])
axes[0].set_title("QQ Plot of Residuals")

# Residual vs Fitted
axes[1].scatter(diag['fitted'], diag['residuals'], alpha=0.5)
axes[1].axhline(y=0, color='r', linestyle='--')
axes[1].set_xlabel("Fitted Values")
axes[1].set_ylabel("Residuals")
```

```
axes[1].set_title("Residuals vs Fitted")

plt.tight_layout()
plt.show()

# 3. Check for outliers
if diag['n_outliers'] > 0:
    print(f"[WARNING] {diag['n_outliers']} potential outliers - investigate these")
```

4.8 Report Generation (report.py)

Purpose Generate publication-quality tables and summaries.

Main Functions

`descriptive_table()`

```
def descriptive_table(
    ds: AnalysisDataset,
    outcomes: Optional[List[str]] = None,
    by_group: Optional[str] = None,
    format_spec: str = ".2f",
) -> pd.DataFrame:
    """
    Generate "Table 1" style descriptive statistics.

    Output format:
        | Outcome | N | Mean (SD) | Median [IQR] | Range |
    """
```

`coefficient_table()`

```
def coefficient_table(
    result: LMMResult,
    format_spec: str = ".3f",
    include_ci: bool = True,
) -> pd.DataFrame:
    """
    Format model coefficients for publication.

    Output format:
        | Term | Estimate (95% CI) | SE | p-value |
    """
```

`results_summary()`

```
def results_summary(
    results: Dict[str, LMMResult],
    fdr_results: Optional[pd.DataFrame] = None,
) -> pd.DataFrame:
    """
    Comprehensive summary across all models.
    Combines model fit statistics with FDR-adjusted p-values.
    """
```

Variance Explained: ICC and R-Squared Measures For longitudinal models, reviewers often want measures of variance explained. The LMMResult includes several complementary metrics:

Intraclass Correlation (ICC)

Already computed in `result['random_effects']['icc']`:

`ICC = sigma^2_u / (sigma^2_u + sigma^2_residual)`

Interpretation: Proportion of total variance attributable to between-subject differences.

R-Squared Measures for Mixed Models

Measure	Formula	Interpretation
Marginal R^2	Var(fixed) / Var(total)	Variance explained by fixed effects only
Conditional R^2	Var(fixed + random) / Var(total)	Variance explained by full model

```
# Access R² measures (when available):
print(result['fit_stats']['r2_marginal'])     # Fixed effects only
print(result['fit_stats']['r2_conditional'])  # Fixed + random effects

# Typical interpretation:
# - Marginal R² = 0.15: Fixed effects explain 15% of variance
# - Conditional R² = 0.65: Full model explains 65% (random effects add 50%)
```

> **Current Status**: ICC is implemented in v0.2.0. Marginal and conditional R^2 (Nakagawa & Schielzeth, 2013) are planned for v0.3.0. When implemented, they will appear in `results_summary()` output automatically.

Recommended Reporting:

For OH/biomedical papers, report: 1. **ICC**: Shows clustering strength (justifies mixed model) 2. **Marginal R^2**: Shows practical effect of predictors 3. Raw variance components (sigma^2_u, sigma^2_residual) for reproducibility

5. Questionnaire Data Support

5.1 Overview

Version 0.3.0 adds comprehensive support for questionnaire data commonly used in occupational health research. The package handles:

Questionnaire	Type	Items	Measurement	Frequency
COPSOQ	Psychosocial	23 dimensions	1-5 Likert -> 0-100	Baseline
MUEQ	Ergonomic risk	6 domains	1-5 Likert -> 0-100	Baseline
ROSA	Office strain	1 total	1-10 ordinal	Baseline
IPAQ	Physical activity	MET-min/week	Continuous	Baseline
OSPAQ	Occupational sitting	% time	Proportions	Baseline
NPRS	Pain rating	0-10 per region	Ordinal	Daily
Workload	Daily demands	4-8 items	1-5 Likert	Daily

5.2 Statistical Considerations by Questionnaire Type

COPSOQ/MUEQ Dimension Scores (Continuous) Dimension scores are computed as mean of constituent items, rescaled 0-100:

```
from oh_stats import compute_composite_score

# Example: COPSOQ Quantitative Demands (3 items)
df['copsoq_quant_demands'] = compute_composite_score(
    df,
    items=['copsoq_q1', 'copsoq_q2', 'copsoq_q3'],
    reverse_items=['copsoq_q2'],  # Item 2 is reverse-coded
    scale_max=5,
    output_scale=(0, 100)
)
```

Statistical treatment: - `OutcomeType.CONTINUOUS` with `TransformType.NONE` - Standard Gaussian LMM applies - Check for ceiling/floor effects in descriptives

ROSA Score (Ordinal, 1-10)

```
# ROSA is a single ordinal score
# Current approach: Treat as continuous in LMM (common practice)
# Future: Ordinal mixed model
```

```
result = fit_lmm(ds, 'rosa_total')  # Works, but interpret with care
```

Statistical treatment: - `OutcomeType.ORDINAL` with `TransformType.NONE` - Gaussian LMM is pragmatic fallback (10-point scale often treated as interval) - Future: Cumulative link mixed models (CLMMs)

IPAQ MET-minutes/week (Right-Skewed Continuous)

```
# IPAQ data is highly right-skewed with possible zeros
# Registry recommends LOG1P transform

from oh_stats import fit_lmm, TransformType
result = fit_lmm(ds, 'ipaq_total_met_min_week', transform=TransformType.LOG1P)
```

Statistical treatment: - `OutcomeType.CONTINUOUS` with `TransformType.LOG1P` - Back-transform coefficients for interpretation - Consider categorized IPAQ (low/moderate/high) for robustness check

OSPAQ Percentages (Proportions)

```
# OSPAQ: % time sitting/standing/walking (compositional data)
# Must convert 0-100% to 0-1 proportions
# LOGIT transform handles bounded nature

result = fit_lmm(ds, 'ospaq_sitting_pct')  # Auto-applies LOGIT
```

Statistical treatment: - `OutcomeType.PROPORTION` with `TransformType.LOGIT` - Epsilon clamping (1e-6) handles 0% and 100% values - Note: Three OSPAQ components sum to ~100%, creating compositional dependency - Analyze one at a time, or use compositional data analysis

NPRS Pain (Ordinal, 0-10, Daily Repeated)

```
# Daily pain ratings are ordinal with many zeros (no pain)
# High zero-inflation is common

from oh_stats import prepare_daily_pain

pain_ds = prepare_daily_pain(profiles, body_regions=['neck', 'shoulder'])
result = fit_lmm(pain_ds, 'nprs_neck')
```

Statistical treatment: - `OutcomeType.ORDINAL` with `TransformType.NONE` - Consider: Binary analysis (pain yes/no) as sensitivity check - Future: Zero-inflated ordinal models

5.3 Multi-Modal Analysis: Combining Sensors and Questionnaires

A key research question is linking subjective reports to objective measurements:

```
from oh_parser import load_profiles
from oh_stats import (
```

```
    prepare_daily_emg,
    prepare_daily_workload,
    align_sensor_questionnaire,
    fit_lmm
)

# Load profiles
profiles = load_profiles("/path/to/profiles")

# Prepare EMG data
emg_ds = prepare_daily_emg(profiles, side="average")

# Prepare daily workload
workload_ds = prepare_daily_workload(profiles)

# Merge on subject x date
merged_ds = align_sensor_questionnaire(emg_ds, workload_ds)

# Now can model: Does perceived workload predict EMG intensity?
# Formula: EMG_intensity ~ workload_composite + (1|subject_id)
result = fit_lmm(
    merged_ds,
    outcome='EMG_intensity.mean_percent_mvc',
    fixed_effects=['workload_composite'],
    day_as_categorical=False  # workload as continuous predictor
)
```

5.4 Baseline Questionnaires as Covariates

Single-instance questionnaires can be used as subject-level covariates:

```
from oh_stats import prepare_baseline_questionnaires, prepare_daily_emg

# Prepare baseline data
baseline_ds = prepare_baseline_questionnaires(profiles, questionnaire_type="copsoq")
emg_ds = prepare_daily_emg(profiles, side="both")

# Merge baseline covariates into EMG data
merged = emg_ds['data'].merge(
    baseline_ds['data'][['subject_id', 'copsoq_quantitative_demands']],
    on='subject_id'
)

# Research question: Do workers with higher psychosocial demands
# show different EMG patterns?
```

6. Statistical Methods Deep Dive

6.1 Linear Mixed Models: Mathematical Foundation

The Model Full LMM specification:

$$\mathbf{Y} = \mathbf{X}\beta + \mathbf{Zu} + \varepsilon$$

Where: - $\mathbf{Y}$: $n \times 1$ vector of observations - $\mathbf{X}$: $n \times p$ fixed effects design matrix - β: $p \times 1$ vector of fixed effect coefficients - $\mathbf{Z}$: $n \times q$ random effects design matrix - $\mathbf{u}$: $q \times 1$ vector of random effects, $\mathbf{u} \sim N(\mathbf{0}, \mathbf{G})$ - ε: $n \times 1$ residual vector, $\varepsilon \sim N(\mathbf{0}, \mathbf{R})$

For Random Intercept Model

$$Y_{ij} = \beta_0 + \beta_1 \text{Day}_{ij} + \beta_2 \text{Side}_{ij} + u_i + \varepsilon_{ij}$$

Variance structure: - Var(u_i) = sigma^2_u (between-subject variance) - Var(e_ij) = sigma^2 (within-subject variance) - Cov(Y_ij, Y_ik) = sigma^2_u for same subject (compound symmetry)

6.2 Variance Transforms

Transform Reference

Transform	Formula	Use When	Status
LOG	$\log(Y)$	Y > 0, right-skewed	[OK] Recommended
LOG1P	$\log(1 + Y)$	Y >= 0 with zeros	[~] Fallback
SQRT	$\sqrt{Y}$	Y >= 0, moderate skew	[OK] Recommended
LOGIT	$\log \frac{Y}{1-Y}$	Y in (0,1), proportions	[OK] **Preferred**
ARCSINE	$\arcsin(\sqrt{Y})$	Legacy only	[X] **Deprecated**

6.3 Multiple Testing Correction

FDR (False Discovery Rate)

$$\text{FDR} = E\left[\frac{V}{R}\right]$$

Benjamini-Hochberg Procedure: 1. Order p-values: p_(1) <= p_(2) <= ... <= p_(m) 2. Find largest k such that p_(k) <= (k/m) * alpha 3. Reject all H_(1), H_(2), ..., H_(k)

FWER (Family-Wise Error Rate) Holm Procedure: 1. Order p-values 2. Compare p_(i) to alpha / (m - i + 1) 3. Reject until first non-rejection, then stop

7. Complete Workflow Examples

7.1 Basic Analysis: Day Effect on EMG Intensity

```
"""
Research Question: Does EMG intensity change over the monitoring period?
"""

from oh_parser import load_profiles
from oh_stats import (
    prepare_daily_emg,
    summarize_outcomes,
    check_normality,
    check_variance,
    fit_lmm,
    pairwise_contrasts,
    residual_diagnostics,
    coefficient_table,
    summarize_lmm_result,
)

# Step 1: Load and Prepare Data
profiles = load_profiles("/path/to/profiles")
ds = prepare_daily_emg(profiles, side="both")

outcome = "EMG_intensity.mean_percent_mvc"

# Step 2: Descriptive Statistics
print("=== Descriptive Statistics ===")
summary = summarize_outcomes(ds, [outcome])
print(summary)

# Step 3: Fit Model
print("\n=== Linear Mixed Model ===")
result = fit_lmm(ds, outcome)

if not result['converged']:
    print(f"WARNING: Model did not converge! {result['warnings']}")
else:
    print(summarize_lmm_result(result))
    print("\nCoefficients:")
    print(coefficient_table(result))

# Step 4: Diagnostics
print("\n=== Diagnostics ===")
diag = residual_diagnostics(result)
print(f"Normality p = {diag['normality_p']:.4f}")
print(f"Outliers detected: {diag['n_outliers']}")

# Step 5: Post-Hoc Comparisons
print("\n=== Post-Hoc Contrasts ===")
contrasts = pairwise_contrasts(result, "day_index", ds)
print(contrasts[["contrast", "estimate", "p_adjusted", "cohens_d"]])
```

7.2 Multi-Outcome Analysis with FDR Correction

```
"""
Research Question: Which EMG metrics change significantly over time?
"""

from oh_stats import (
    prepare_daily_emg,
    fit_all_outcomes,
    apply_fdr,
    results_summary,
    list_outcomes,
    OutcomeType,
    check_variance,
)

# Load Data
profiles = load_profiles("/path/to/profiles")
ds = prepare_daily_emg(profiles, side="both")

# Pre-modeling Checks
continuous_outcomes = list_outcomes(outcome_type=OutcomeType.CONTINUOUS)
variance_check = check_variance(ds, continuous_outcomes)
degenerate = variance_check[variance_check["is_degenerate"]]["outcome"].tolist()
print(f"Degenerate outcomes (excluded): {degenerate}")

# Fit All Models
results = fit_all_outcomes(
    ds,
    outcome_type=OutcomeType.CONTINUOUS,
    skip_degenerate=True,
)
print(f"\nFitted {len(results)} models")

# Apply FDR Correction
fdr_df = apply_fdr(results, alpha=0.05)
print("\n=== FDR-Corrected Results ===")
print(fdr_df)

# Summary Table
summary = results_summary(results, fdr_df)
print("\n=== Complete Summary ===")
print(summary)
```

7.3 Questionnaire Analysis Workflow (v0.3.0+)

```
"""
Research Question: Do COPSOQ psychosocial factors relate to EMG activity?
"""

from oh_parser import load_profiles
from oh_stats import (
```

```
    prepare_daily_emg,
    prepare_baseline_questionnaires,
    prepare_daily_pain,
    align_sensor_questionnaire,
    get_questionnaire_outcomes,
    fit_lmm,
    apply_fdr,
    summarize_outcomes,
)

# Load profiles
profiles = load_profiles("E:/OH_profiles")

# Step 1: Prepare questionnaire data
baseline_ds = prepare_baseline_questionnaires(profiles, questionnaire_type="copsoq")
pain_ds = prepare_daily_pain(profiles)

# Step 2: Prepare EMG data
emg_ds = prepare_daily_emg(profiles, side="average")

# Step 3: Describe questionnaire distributions
print("=== COPSOQ Baseline Summary ===")
print(summarize_outcomes(baseline_ds))

print("\n=== Daily Pain Summary ===")
print(summarize_outcomes(pain_ds))

# Step 4: Merge sensor + questionnaire for multi-modal analysis
merged_ds = align_sensor_questionnaire(emg_ds, pain_ds)

# Step 5: Model pain as function of day (trajectory analysis)
pain_result = fit_lmm(pain_ds, 'nprs_neck')
print(f"\nNeck pain trajectory: p = {pain_result['fit_stats']['lrt_pvalue']:.4f}")

# Step 6: Model EMG controlling for daily pain
# (Add pain as time-varying covariate)
print("\n=== EMG with Pain Covariate ===")
emg_pain_result = fit_lmm(
    merged_ds,
    outcome='EMG_intensity.mean_percent_mvc',
    fixed_effects=['day_index', 'nprs_neck'],
    day_as_categorical=False
)
```

8. Edge Cases and Error Handling

8.1 Missing Data

```
# Check missingness before modeling
miss = missingness_report(ds)
print(f"Total missing: {miss['total_missing'].iloc[0]} cells")

# If >10% missing, investigate patterns
if miss['pct_missing'].iloc[0] > 10:
    print("Warning: High missingness - check if MAR is plausible")
```

8.2 Non-Convergence

```
result = fit_lmm(ds, outcome)

if not result['converged']:
    print("Model did not converge!")
    print(f"Warnings: {result['warnings']}")

    # Try simpler model
    result_simple = fit_lmm(ds, outcome, include_side=False)

    # Or try transformation
    result_log = fit_lmm(ds, outcome, transform=TransformType.LOG)
```

8.3 Degenerate Outcomes

```
variance_check = check_variance(ds, [outcome])
if variance_check['is_degenerate'].iloc[0]:
    print(f"Outcome {outcome} is degenerate")
    print("This outcome cannot be meaningfully modeled.")
```

8.4 Small Sample Sizes

```
result = fit_lmm(ds, outcome)

if result['n_groups'] < 30:
    print(f"Warning: Only {result['n_groups']} subjects - results may be unstable")
```

[!] Model-Aware Sample Size Considerations

The "<30 clusters" heuristic is a rough guideline. Actual stability depends on:

1. **Number of clusters (subjects)**: More important than total observations
2. **Fixed-effect complexity**: More coefficients require more clusters
3. **Observations per cluster**: Sparse designs amplify instability
4. **Missingness patterns**: Systematic dropout is worse than random

When clusters are limited (<30 subjects): - Prefer simpler models (fewer fixed effects) - Consider wider confidence intervals or bootstrap CIs - Interpret random effect

variance estimates with caution (often imprecise) - Report sensitivity analyses with different model specifications

There is no magic threshold - use judgment based on design complexity.

9. Best Practices

9.1 Pre-Analysis Checklist

1. **Check data quality**
   - Run `missingness_report()`
   - Verify sample sizes are adequate
2. **Examine distributions**
   - Run `summarize_outcomes()` for all outcomes
   - Run `check_normality()` to identify skewness
   - Run `check_variance()` to identify degenerate variables
3. **Document analysis plan**
   - Specify primary vs. secondary outcomes
   - Pre-specify alpha level and correction method
   - Justify model specification

9.2 Working with TypedDicts

```
# Access data using dictionary syntax
print(ds['data'].shape)
print(result['fit_stats']['aic'])
print(diag['n_outliers'])

# Use helper functions for common operations
description = describe_dataset(ds)
summary = summarize_lmm_result(result)
diag_summary = summarize_diagnostics(diag)

# Factory functions ensure proper structure
ds = create_analysis_dataset(data=df, outcome_vars=outcomes, ...)
result = create_lmm_result(outcome=name, model=model, ...)
```

9.3 Reporting Results

Example results paragraph: > "EMG mean %MVC was analyzed using linear mixed models with subjects as random intercepts, day (categorical) and side as fixed effects. The ICC was 0.50, indicating substantial between-subject variation. After FDR correction (alpha = 0.05), day showed a significant overall association with mean %MVC (p_adj = 0.03). Post-hoc contrasts (Holm-adjusted) revealed significantly lower activation on Day 4 compared to Day 1 (Delta = -1.93 %MVC, 95% CI [-3.58, -0.28], Cohen's d = 0.28, p_adj = 0.04)."

10. Glossary

Term	Definition
AIC	Akaike Information Criterion. Lower = better fit.
AnalysisDataset	TypedDict container for analysis-ready data.
Cohen's d	Standardized effect size: (M1 - M2) / SD_pooled.
ContrastResult	TypedDict for post-hoc comparison output.
DiagnosticsResult	TypedDict for model diagnostics output.
discover_sensors()	Function to find available sensors and metrics in profiles.
discover_questionnaires()	Function to find available questionnaire domains and fields.
FDR	False Discovery Rate. Expected proportion of false positives.
get_profile_summary()	Function to generate human-readable data overview.
ICC	Intraclass Correlation. Between-subject variance proportion.
LMMResult	TypedDict for linear mixed model output.
OutcomeInfo	TypedDict for outcome variable metadata.
prepare_sensor_data()	Generic function to prepare ANY sensor data for analysis.
TypedDict	Python dict with type hints (function-based alternative to dataclass).
COPSOQ	Copenhagen Psychosocial Questionnaire (work stress dimensions).
MUEQ	Musculoskeletal-Utrecht Ergonomic Questionnaire.
ROSA	Rapid Office Strain Assessment (1-10 score).
IPAQ	International Physical Activity Questionnaire (MET-min/week).
OSPAQ	Occupational Sitting and Physical Activity Questionnaire.
NPRS	Numeric Pain Rating Scale (0-10).

Quick Reference Card

```python
# ===== MINIMAL WORKFLOW =====

from oh_parser import load_profiles
from oh_stats import (
    # Discovery
    get_profile_summary,
    discover_sensors,
    discover_questionnaires,
    # Preparation
    prepare_sensor_data,
    prepare_daily_emg,
    # Analysis
    summarize_outcomes,
    fit_all_outcomes,
    apply_fdr,
    results_summary,
)

# 1. Load & Discover
```

```
profiles = load_profiles("/path/to/data")
print(get_profile_summary(profiles))  # See what's available

# 2. Prepare (choose your approach)
# Option A: EMG convenience wrapper
ds = prepare_daily_emg(profiles, side="both")

# Option B: Generic for any sensor
ds = prepare_sensor_data(
    profiles,
    sensor="heart_rate",
    base_path="sensor_metrics.heart_rate",
    level_names=["date"],
    value_paths=["HR_BPM_stats.*"]
)

# 3. Describe
summary = summarize_outcomes(ds)
print(summary)

# 4. Model
results = fit_all_outcomes(ds, skip_degenerate=True)

# 5. Correct
fdr = apply_fdr(results)
print(fdr[fdr['significant']])

# 6. Report
report = results_summary(results, fdr)
print(report)
```

11. Architecture: Generic Data-Type Based Analysis

The `oh_stats` architecture uses a **generic, data-type based** approach that cleanly separates data extraction from statistical analysis:

Separation of Concerns

Layer	Package	Responsibility
Extraction	`oh_parser`	Extract data from ANY sensor using `extract_nested()`, `extract_flat()`
Discovery	`oh_stats`	Inspect profiles to see what data is available
Preparation	`oh_stats`	Transform extracted data into `AnalysisDataset` format
Analysis	`oh_stats`	Statistical analysis based on **data type** (continuous, ordinal, etc.)

Why Generic?

1. **New sensors don't require new functions**: Add heart rate? Just use `prepare_sensor_data()` with appropriate paths
2. **Consistent API**: Same workflow for EMG, heart rate, noise, accelerometer, etc.
3. **Data type drives analysis**: A continuous metric from EMG uses the same statistical model as a continuous metric from heart rate
4. **Registry for customization**: Use `register_outcome()` to specify data type and transform for any new metric

Example: Analyzing a New Sensor

```
from oh_parser import load_profiles
from oh_stats import (
    discover_sensors,
    prepare_sensor_data,
    register_outcome,
    OutcomeType,
    TransformType,
    fit_lmm
)

# 1. Load and discover
profiles = load_profiles("/path/to/data")
sensors = discover_sensors(profiles)
print(sensors['heart_rate'])  # See what metrics are available

# 2. Prepare data (generic function works for any sensor)
ds = prepare_sensor_data(
    profiles,
    sensor="heart_rate",
    base_path="sensor_metrics.heart_rate",
    level_names=["date"],
    value_paths=["HR_BPM_stats.*"]
```

```
)

# 3. Register outcome type (tells oh_stats how to analyze it)
register_outcome(
    name="HR_BPM_stats.mean",
    outcome_type=OutcomeType.CONTINUOUS,
    transform=TransformType.NONE,
    description="Mean heart rate in BPM"
)

# 4. Analyze using the same LMM as any continuous outcome
result = fit_lmm(ds, "HR_BPM_stats.mean")
```

Future Extensions

Feature	Current Status	Future Direction
Count models	LOG1P + Gaussian LMM	Poisson/NB GLMMs
Ordinal models	Stub only	Ordered logistic (Bayesian)
Beta regression	Logit transform	`statsmodels` beta regression
Multi-channel	Subject random intercept	Subject -> Day -> Channel hierarchy

References

- Statsmodels MixedLM: https://www.statsmodels.org/stable/mixed_linear.html
- Benjamini & Hochberg (1995): Controlling the False Discovery Rate. *JRSS-B* 57(1):289-300
- Holm (1979): A Simple Sequentially Rejective Multiple Test Procedure. *Scand J Stat* 6(2):65-70
- Warton & Hui (2011): The arcsine is asinine. *Ecology* 92(1):3-10
- Nakagawa & Schielzeth (2013): A general and simple method for obtaining R^2 from GLMMs. *Methods Ecol Evol* 4(2):133-142

Document generated for oh_stats v0.3.0 Last updated: January 2026 Architecture: Function-based with TypedDicts