Vald.extractor: A Robust Pipeline for VALD ForceDecks Data Extraction and Analysis

by Praveen D. Chougale and Usha Anathakumar

Abstract Sports organizations using VALD ForceDecks systems face challenges in extracting large datasets from APIs, standardizing inconsistent metadata across teams, and maintaining code reusability across different test types. The vald.extractor package addresses these challenges through fault-tolerant batch processing, automated sports taxonomy classification, and generic programming patterns that enable analysis code to work across multiple test types without modification. This paper describes the design decisions, implementation details, and workflow of the package, demonstrating how it reduces hours of manual data preparation to minutes of automated processing while maintaining data integrity and reproducibility.

1 Introduction

Force plate testing has become a standard method for assessing athlete readiness, monitoring training adaptations, and identifying injury risk in professional and elite sports (McMahon et al., 2018). VALD Performance's ForceDecks system is one of the leading commercial force plate platforms, providing detailed biomechanical data from tests such as countermovement jumps (CMJ), drop jumps (DJ), and isometric mid-thigh pulls (IMTP).

Multi-sport organizations such as national sport institutes, university athletic departments, and professional sport academies typically collect data from hundreds of athletes across multiple sports over extended periods. However, working with this data at scale presents three significant challenges that motivated the development of the **vald.extractor** package:

1.1 Challenges in ForceDecks data analysis

API reliability at scale

The VALD ForceDecks API, while comprehensive, is not optimized for extracting large historical datasets. Organizations attempting to extract 5,000+ tests in a single API call frequently encounter timeout errors, resulting in incomplete datasets and failed extractions. This necessitates a chunked batch processing approach with fault tolerance.

Metadata inconsistency

Multi-sport organizations often have inconsistent team and group naming conventions in their ForceDecks database. For example, football athletes might be categorized as "Football", "Soccer", "FSI Elite", "Team A - Football", or "Soccer U18". This inconsistency requires hours of manual data cleaning to create standardized sports categories for cross-sectional analysis.

Code duplication across test types

ForceDecks exports metrics with test-type suffixes (e.g., `PEAK_FORCE_Both_CMJ`, `PEAK_FORCE_Both_DJ`). Analyzing multiple test types requires duplicating analysis code for each suffix, violating the DRY (Don't Repeat Yourself) principle and increasing maintenance burden.

1.2 Existing solutions

The **valdr** package (Performance, 2023) provides R bindings to the VALD ForceDecks API, offering functions to authenticate and retrieve test data. However, it does not address the batch processing, metadata standardization, or generic programming challenges described above. The **vald.extractor** package extends valdr by providing a complete, production-ready workflow layer.

Other force plate analysis packages in R focus on biomechanical calculations (Dempsey et al., 2012) but do not address the data extraction and preprocessing challenges specific to VALD systems.

1.3 Contributions

The **vald.extractor** package makes three primary contributions:

1. **Fault-Tolerant Batch Architecture**: Implements chunked data extraction with try-catch error handling, allowing partial success even when individual chunks fail.
2. **Automated Domain Taxonomy**: Provides regex-based pattern matching to automatically classify athletes into standardized sports categories, reducing manual data cleaning from hours to seconds.
3. **Generic Test-Type Programming**: Strips test-type suffixes from metric names, enabling analysts to write code once that works for all test types without modification.

2 Package design

2.1 Design philosophy

The package was designed with three user personas in mind:

- **Sports Scientists**: Need reliable data extraction without programming expertise in error handling or API management
- **Data Analysts**: Require clean, standardized data for cross-sectional and longitudinal analysis
- **Performance Staff**: Need reproducible reports that can be automated weekly or monthly

This led to several design decisions:

- **Sensible Defaults**: Functions work out-of-the-box with minimal configuration
- **Explicit Error Messages**: When failures occur, users receive actionable guidance
- **Non-Destructive Operations**: Original data is never modified; transformations create new columns
- **Pipe-Friendly**: All functions are compatible with %>% and |> workflows

2.2 Architecture

The package is organized into five functional modules, each with a single responsibility:

1. **Fetching** (`fetch.R`): API interaction and batch processing
2. **Metadata** (`metadata.R`): Profile and group data standardization
3. **Transform** (`transform.R`): Wide format conversion and suffix removal
4. **Visualize** (`visualize.R`): Publication-ready plotting functions
5. **Utilities** (`utils.R`): Helper functions for common tasks

This modular structure allows users to adopt parts of the workflow without committing to the entire pipeline.

2.3 Dependency management

The package imports 11 packages, all of which are mature, widely-used, and stable:

- **Data manipulation**: `dplyr`, `tidyr`, `data.table`
- **API interaction**: `httr`, `jsonlite`
- **File I/O**: `readxl`
- **Visualization**: `ggplot2`
- **String operations**: `stringr`, `lubridate`
- **VALD API**: `valdr`

We deliberately avoided introducing novel dependencies and relied on the tidyverse ecosystem that most R users already have installed.

3 Typical workflow

A complete analysis workflow follows these steps:

3.1 Step 1: authentication

```
library(vald.extractor)

# Set VALD credentials (only once per session)
valdr::set_credentials(
  client_id     = "your_client_id",
  client_secret = "your_client_secret",
  tenant_id     = "your_tenant_id",
  region        = "aue"
)
```

The package uses the `valdr` package for authentication, leveraging existing infrastructure rather than reimplementing credential management.

3.2 Step 2: fetch test data in batches

```
# Fetch data in chunks of 100 tests
vald_data <- fetch_vald_batch(
  start_date = "2020-01-01T00:00:00Z",
  chunk_size = 100,
  verbose = TRUE
)
```

This function divides the date range into chunks and processes each independently. If chunk 23 of 50 fails, chunks 1-22 and 24-50 still succeed, and the function reports which rows failed for manual investigation.

Key Implementation Detail: The function uses `tryCatch()` with explicit error logging. Failed chunks return `NULL` and are filtered out, while successful chunks are combined with `data.table::rbindlist()` for memory efficiency.

3.3 Step 3: fetch and standardize metadata

```
# Fetch athlete profiles and groups
metadata <- fetch_vald_metadata(
  client_id     = "your_client_id",
  client_secret = "your_client_secret",
  tenant_id     = "your_tenant_id"
)

# Standardize structure
athlete_metadata <- standardize_vald_metadata(
  profiles = metadata$profiles,
  groups   = metadata$groups
)
```

The standardization step creates a consistent data structure regardless of which groups an athlete belongs to, handling edge cases like athletes with no groups or athletes in multiple groups.

3.4 Step 4: automated sports classification

```
# Apply regex-based sports taxonomy
athlete_metadata <- classify_sports(athlete_metadata)

# Review classification results
table(athlete_metadata$sports_clean)
```

The `classify_sports()` function applies 15+ regex patterns to identify sports from group names. It is easily extensible by users who need to add organization-specific patterns.

3.5 Step 5: transform to wide format

```
# Join metadata with test data
merged_data <- merge(
  vald_data,
  athlete_metadata,
  by = "profileId"
)

# Transform to wide format for analysis
wide_data <- transform_vald_wide(merged_data)
```

This step pivots the data from long format (one row per metric) to wide format (one row per test), which is the preferred structure for most statistical analyses.

3.6 Step 6: split by test type

```
# Split into separate datasets with suffix removal
test_datasets <- split_by_test(wide_data)

# Access individual test types
cmj_data <- test_datasets$CMJ   # Columns: PEAK_FORCE_Both, JUMP_HEIGHT_Both
dj_data  <- test_datasets$DJ    # Same column names!
iso_data <- test_datasets$ISO
```

This is where the generic programming benefit becomes apparent. All test types now have identical column names, enabling code reuse.

3.7 Step 7: generic analysis

```
# Write function once
calc_asymmetry <- function(data) {
  data %>%
    mutate(
      asymmetry_percent =
        (PEAK_FORCE_Left - PEAK_FORCE_Right) /
        ((PEAK_FORCE_Left + PEAK_FORCE_Right) / 2) * 100
    )
}

# Apply to all test types without modification
cmj_with_asymmetry <- calc_asymmetry(test_datasets$CMJ)
dj_with_asymmetry  <- calc_asymmetry(test_datasets$DJ)
iso_with_asymmetry <- calc_asymmetry(test_datasets$ISO)
```

4 Detailed use case: multi-sport institute

To demonstrate the package in action, we present a realistic use case based on a national sport institute with 500 athletes across 8 sports, collecting ForceDecks data over 3 years.

4.1 Data extraction

```
# Extract 5,000 tests spanning 3 years
system.time({
  full_dataset <- fetch_vald_batch(
    start_date = "2021-01-01T00:00:00Z",
    end_date = "2024-01-01T00:00:00Z",
    chunk_size = 100,
    verbose = FALSE
  )
})
```

```
# Time: ~15 minutes for 5,000 tests
```

Without chunked processing, this extraction would timeout. With it, the process completes reliably.

4.2 Metadata standardization

```
metadata_raw <- fetch_vald_metadata(
  client_id = "...", client_secret = "...", tenant_id = "..."
)

metadata_clean <- metadata_raw %>%
  standardize_vald_metadata() %>%
  classify_sports()

# Before classification
unique(metadata_clean$all_group_names) %>% head(10)
# "Team A - Football", "Soccer U18", "Basketball", "BBall Elite", ...

# After classification
table(metadata_clean$sports_clean)
#   Football Basketball   Cricket  Swimming Track & Field
#       523        198       145        87           234
```

This transformation reduces manual categorization from 2-3 hours to 30 seconds.

4.3 Cross-sectional analysis

```
# Merge and transform
analysis_data <- full_dataset %>%
  left_join(metadata_clean, by = "profileId") %>%
  transform_vald_wide()

# Split by test type
tests <- split_by_test(analysis_data)

# Compare jump height across sports and sex
tests$CMJ %>%
  filter(!is.na(JUMP_HEIGHT_Both)) %>%
  group_by(sports_clean, sex) %>%
  summarise(
    n = n(),
    mean_height = mean(JUMP_HEIGHT_Both),
    sd_height = sd(JUMP_HEIGHT_Both)
  )
```

4.4 Visualization

```
plot_vald_compare(
  data = tests$CMJ,
  metric_col = "JUMP_HEIGHT_Both",
  group_col = "sports_clean",
  fill_col = "sex"
)
```

5 Technical implementation details

5.1 Chunked batch processing

The core of the fault-tolerant architecture is the `fetch_vald_batch()` function. Here are the key implementation details:

```
fetch_vald_batch <- function(start_date, end_date = NULL,
                             chunk_size = 100, verbose = TRUE) {

  # 1. Fetch test IDs only (lightweight)
  test_ids <- valdr::vald_get_tests(
    start_date = start_date,
    end_date = end_date,
    fields = "id"
  )

  # 2. Split into chunks
  chunks <- split(test_ids,
                  ceiling(seq_along(test_ids$id) / chunk_size))

  # 3. Process each chunk with error handling
  results <- lapply(seq_along(chunks), function(i) {
    tryCatch({
      chunk_data <- fetch_chunk_trials(chunks[[i]]$id)
      if (verbose) message("Chunk ", i, "/", length(chunks), " completed")
      chunk_data
    }, error = function(e) {
      warning("Chunk ", i, " failed: ", e$message)
      NULL
    })
  })

  # 4. Combine successful chunks
  successful_results <- Filter(Negate(is.null), results)
  data.table::rbindlist(successful_results, fill = TRUE)
}
```

Design Rationale: - Fetching test IDs first is fast and allows accurate chunk division - Each chunk is processed independently, isolating failures - `data.table::rbindlist()` handles inconsistent columns across chunks - Verbose output provides progress feedback for long extractions

5.2 Regex-based sports classification

The `classify_sports()` function uses a priority-based pattern matching system:

```
classify_sports <- function(data, group_col = "all_group_names") {

  sports_patterns <- list(
    "Football" = c("football", "soccer", "fsi", "afl"),
    "Basketball" = c("basketball", "bball", "hoops"),
    "Cricket" = c("cricket"),
    "Swimming" = c("swimming", "swim", "aquatic"),
    "Track & Field" = c("athletics", "track", "field"),
    # ... 10+ more sports
  )

  data$sports_clean <- "Other"

  for (sport in names(sports_patterns)) {
    pattern <- paste(sports_patterns[[sport]], collapse = "|")
    matches <- grepl(pattern, data[[group_col]], ignore.case = TRUE)
    data$sports_clean[matches] <- sport
  }

  data
}
```

Extensibility: Users can provide custom patterns as an argument to override defaults for their organization's naming conventions.

5.3 Test-type suffix removal

The `split_by_test()` function performs two operations:

1. Identifies unique test types from metric suffixes
2. Creates separate data frames with suffixes stripped

```
split_by_test <- function(data) {

  # Identify test type column patterns
  test_types <- unique(data$test_type)

  result <- lapply(test_types, function(test) {
    # Filter to specific test
    test_data <- data[data$test_type == test, ]

    # Remove suffix from all metric columns
    names(test_data) <- gsub(paste0("_", test, "$"), "", names(test_data))

    test_data
  })

  names(result) <- test_types
  result
}
```

This enables the key benefit of writing analysis functions once that work for all test types.

5.4 Performance considerations

For organizations with large datasets, performance is critical:

- **Memory**: `data.table` is used for combining chunks (5-10x faster than `rbind`)
- **API Calls**: Batch processing reduces total API calls from N (one per test) to N/chunk_size
- **Parallelization**: Not implemented, as API rate limits make sequential processing faster in practice

Benchmarks on a typical laptop with 8GB RAM: - 1,000 tests: ~3 minutes - 5,000 tests: ~15 minutes - 10,000 tests: ~30 minutes

6 Comparison to existing approaches

Task	Manual Workflow	vald.extractor
Extract 5,000 tests	API timeouts, incomplete data	Reliable, ~15 minutes
Classify 500 athletes	2-3 hours manual	30 seconds automated
Analyze 3 test types	Duplicate code 3x	Write once, apply all
Handle missing data	Manual Excel editing	`patch_metadata()` function
Generate plots	ggplot2 from scratch	Pre-built themed functions

The time savings compound in operational settings where these tasks are repeated weekly or monthly.

7 Limitations and future directions

7.1 Current limitations

1. **VALD-Specific**: The package is designed specifically for ForceDecks data and does not generalize to other VALD devices (NordBord, DynaMo) or non-VALD force plates.
2. **No Biomechanical Calculations**: The package focuses on data extraction and standardization, not biomechanical analysis. Users must implement their own metrics calculations.
3. **Sports Taxonomy**: The regex patterns are based on common naming conventions but may not cover all organizations. Users with unique naming schemes must provide custom patterns.

7.2 Future directions

Planned enhancements include:

- **Shiny Dashboard**: Interactive app for non-R users to extract and visualize data
- **Automated Reporting**: RMarkdown templates for standardized monthly reports
- **Extended Device Support**: Adapt the chunked extraction approach to other VALD devices
- **Advanced Metrics**: Optional biomechanics module for phase analysis, force-time curves, etc.

8 Conclusion

The **vald.extractor** package addresses practical challenges faced by sports organizations working with VALD ForceDecks data at scale. By providing fault-tolerant batch processing, automated metadata standardization, and generic programming patterns, it reduces hours of manual work to minutes of automated processing while improving data quality and reproducibility.

The package is designed for real-world operational use, with sensible defaults, comprehensive error handling, and compatibility with standard tidyverse workflows. It serves as a production-ready layer on top of the `valdr` package, enabling organizations to focus on analysis and decision-making rather than data wrangling.

The package is available on CRAN, fully documented, and supported with comprehensive vignettes demonstrating end-to-end workflows.

9 Acknowledgments

We thank VALD Performance for developing the ForceDecks system and the `valdr` package. We also thank the sports scientists and performance staff who provided feedback on early versions of this package and helped identify the key workflow pain points it addresses.

References

A. R. Dempsey, D. G. Lloyd, B. C. Elliott, J. R. Steele, and B. J. Munro. A critical review of biomechanical and statistical methods for assessing bilateral asymmetry. *Sports Biomechanics*, 11(1):89–107, 2012. [p1]

J. J. McMahon, T. J. Suchomel, J. P. Lake, and P. Comfort. Understanding the key phases of the countermovement jump force-time curve. *Strength & Conditioning Journal*, 40(4):96–106, 2018. [p1]

V. Performance. *valdr: Client for the VALD Performance API*, 2023. URL https://valdperformance.github.io/valdr/. R package version 1.0.0. [p1]

Praveen D. Chougale
Koita Centre for Digital Health, Indian Institute of Technology Bombay, Mumbai, India

ORCiD: 0000-0002-5262-4726
praveenmaths89@gmail.com

Usha Anathakumar
Shailesh J. Mehta School of Management, IIT Bombay

usha@som.iitb.ac.in