Dynamic Column Names in R: Seven Approaches Compared

Ronald G. Thomas

2026-02-16

Introduction

While reviewing some production R code recently, the following pattern appeared:

```
data |>
  mutate(
    !!paste0(measure, "_bl") := baseline_value,
    !!paste0(measure, "_cng") := current - baseline_value
  )
```

The `!!` and `:=` operators were unfamiliar. After investigation, they proved to be part of R's tidy evaluation system: a powerful but often misunderstood feature for programmatic data manipulation.

This exploration led to cataloging seven different approaches to creating columns with dynamic names in R:

1. Base R using `[[` assignment
2. Base R using `do.call()` and `setNames()`
3. The "classic" tidyverse approach with `!!` and `:=`
4. The modern tidyverse glue-style syntax

5. The rlang expression splicing pattern
6. The data.table approach
7. The collapse package approach

The Problem

Consider a function that calculates change scores for any cognitive measure. Given a dataframe with columns `rid` (subject ID), `vis` (visit), and a measure like `mmse`, we wish to create two new columns:

- `mmse_bl`: the baseline value for each subject
- `mmse_cng`: the change from baseline at each visit

The column names must be constructed dynamically because the function should work for any measure (`mmse`, `adas13`, `cdr`, etc.).

Here is our sample data:

```
library(dplyr)

df <- tibble(
  rid = rep(c("001", "002"), each = 3),
  vis = rep(c("bl", "m06", "m12"), 2),
  mmse = c(28, 27, 25, 30, 29, 28)
)

df
```

```
# A tibble: 6 x 3
  rid   vis    mmse
  <chr> <chr> <dbl>
1 001   bl       28
2 001   m06      27
3 001   m12      25
4 002   bl       30
5 002   m06      29
6 002   m12      28
```

Approach 1: Base R with `[[` Assignment

The most straightforward approach uses base R's `[[` assignment operator:

```r
calculate_change_base <- function(data, measure) {
  bl_col <- paste0(measure, "_bl")
  cng_col <- paste0(measure, "_cng")

  result <- data

  for (id in unique(data$rid)) {
    idx <- data$rid == id
    baseline <- data[[measure]][idx & data$vis == "bl"][1]
    result[[bl_col]][idx] <- baseline
    result[[cng_col]][idx] <- data[[measure]][idx] - baseline
  }

  result
}

calculate_change_base(df, "mmse")
```

```
# A tibble: 6 x 5
  rid   vis    mmse mmse_bl mmse_cng
  <chr> <chr> <dbl>   <dbl>    <dbl>
1 001   bl       28      28        0
2 001   m06      27      28       -1
3 001   m12      25      28       -3
4 002   bl       30      30        0
5 002   m06      29      30       -1
6 002   m12      28      30       -2
```

Discussion

How it works: The `[[` operator accepts character strings for column names, unlike `$` which requires literal names. Writing `df[["mmse"]]` is equivalent to `df$mmse`, but the former allows `df[[variable]]` where `variable` contains the column name.

Advantages:

- No dependencies beyond base R
- Syntax is familiar to programmers from other languages
- Explicit and easy to debug

Disadvantages:

- Verbose, especially with grouped operations
- Requires explicit loops for by-group calculations
- Mutable state pattern (`result` is modified in place)
- Does not integrate with dplyr pipelines
- Performance degrades with many groups

When to use: Small scripts with no package dependencies, or when interfacing with code from other languages where this pattern is standard.

Approach 2: Base R with `do.call()` and `setNames()`

A more functional base R approach avoids explicit loops:

```
calculate_change_docall <- function(data, measure) {
  bl_col <- paste0(measure, "_bl")
  cng_col <- paste0(measure, "_cng")

  baselines <- tapply(
    data[[measure]][data$vis == "bl"],
    data$rid[data$vis == "bl"],
    FUN = `[`, 1
  )

  bl_values <- baselines[data$rid]
  cng_values <- data[[measure]] - bl_values

  new_cols <- setNames(
    list(as.numeric(bl_values), as.numeric(cng_values)),
    c(bl_col, cng_col)
  )

  do.call(cbind, c(list(data), new_cols))
}

calculate_change_docall(df, "mmse")
```

```
  rid vis mmse mmse_bl mmse_cng
1 001  bl   28      28        0
2 001 m06   27      28       -1
3 001 m12   25      28       -3
4 002  bl   30      30        0
5 002 m06   29      30       -1
6 002 m12   28      30       -2
```

Discussion

How it works: `setNames()` creates a named list where the names come from a character vector. `do.call(cbind, ...)` then binds these columns to the original dataframe. The `tapply()` function computes grouped summaries without explicit loops.

Advantages:

- No external dependencies
- Functional style (no mutable state)
- Vectorized operations for better performance

Disadvantages:

- Dense, less readable syntax
- `tapply()` returns an array requiring careful indexing
- Type coercion issues (note the `as.numeric()` calls)
- Returns a `data.frame`, not a `tibble`

When to use: Package development where minimizing dependencies is critical, or performance-sensitive code where the overhead of dplyr is unacceptable.

Figure 1: A wooden filing drawer pulled open to reveal several folders, each labeled in a different handwriting style: the same data, named differently across approaches.

Approach 3: Classic Tidyverse with `!!` and `:=`

The tidyverse introduced tidy evaluation to handle dynamic column names. Two operators are central:

- `:=` (the "walrus" operator): Allows the left-hand side of an assignment to be evaluated
- `!!` (bang-bang): Unquotes an expression, forcing immediate evaluation

```r
calculate_change_classic <- function(data, measure) {
  bl_col <- paste0(measure, "_bl")
  cng_col <- paste0(measure, "_cng")

  data |>
    group_by(rid) |>
    mutate(
      !!bl_col := .data[[measure]][vis == "bl"][1],
      !!cng_col := .data[[measure]] - .data[[bl_col]]
    ) |>
    ungroup()
}

calculate_change_classic(df, "mmse")
```

```
# A tibble: 6 x 5
  rid   vis    mmse mmse_bl mmse_cng
  <chr> <chr> <dbl>   <dbl>    <dbl>
1 001   bl       28      28        0
2 001   m06      27      28       -1
3 001   m12      25      28       -3
4 002   bl       30      30        0
5 002   m06      29      30       -1
6 002   m12      28      30       -2
```

Discussion

How it works: Standard `mutate()` syntax like `mutate(new_col = value)` treats `new_col` as a literal name. The `=` operator does not evaluate its left-hand side. The `:=` operator changes this behavior. When `mutate(!!bl_col := value)` is written, the `!!` forces `bl_col` to be evaluated (yielding `"mmse_bl"`), and `:=` uses that string as the column name.

The `.data` pronoun explicitly references columns in the dataframe, avoiding ambiguity between dataframe columns and local variables.

Advantages:

- Integrates directly into dplyr pipelines
- Grouped operations are trivial
- Declarative, readable intent (once the syntax is learned)

Disadvantages:

- Unfamiliar syntax for newcomers (`!!` and `:=` are not standard R)
- Requires understanding tidy evaluation concepts
- The `rlang` package must be available (loaded with dplyr)

When to use: Legacy tidyverse code (pre-dplyr 1.0), or when the full power of quasiquotation is needed for complex metaprogramming.

Approach 4: Modern Tidyverse with Glue Syntax

Starting with dplyr 1.0 (June 2020), a cleaner syntax emerged using glue-style interpolation:

```
calculate_change_modern <- function(data, measure) {
  data |>
    group_by(rid) |>
    mutate(
      "{measure}_bl" := .data[[measure]][vis == "bl"][1],
      "{measure}_cng" := .data[[measure]] - .data[[paste0(measure, "_bl")]]
    ) |>
    ungroup()
}

calculate_change_modern(df, "mmse")
```

```
# A tibble: 6 x 5
  rid   vis    mmse mmse_bl mmse_cng
  <chr> <chr> <dbl>   <dbl>    <dbl>
1 001   bl       28      28        0
2 001   m06      27      28       -1
3 001   m12      25      28       -3
4 002   bl       30      30        0
5 002   m06      29      30       -1
6 002   m12      28      30       -2
```

Discussion

How it works: The string `"{measure}_bl"` is interpolated like `glue::glue()`, substituting the value of `measure` directly. This occurs at the level of the column name, with `:=` still required to enable left-hand side evaluation.

Advantages:

- Most readable of all tidyverse approaches
- Reduces cognitive load (no need to remember `!!` semantics)
- Consistent with glue syntax used elsewhere in the tidyverse

Disadvantages:

- Requires dplyr >= 1.0
- Still requires `:=` (cannot use `=`)

- Less flexible than `!!` for complex quasiquotation

When to use: New tidyverse code. This is the current recommended approach for dynamic column names in dplyr.

Approach 5: rlang Expression Splicing

For more explicit metaprogramming, rlang provides `expr()` for building expressions and `!!!` for splicing them into function calls:

```
library(rlang)

calculate_change_rlang <- function(data, measure) {
  bl_col <- paste0(measure, "_bl")
  cng_col <- paste0(measure, "_cng")

  exprs <- list(
    expr(.data[[!!measure]][vis == "bl"][1]),
    expr(.data[[!!measure]] - .data[[!!bl_col]])
  )
  names(exprs) <- c(bl_col, cng_col)

  data |>
    group_by(rid) |>
    mutate(!!!exprs) |>
    ungroup()
}

calculate_change_rlang(df, "mmse")
```

```
# A tibble: 6 x 5
  rid   vis    mmse mmse_bl mmse_cng
  <chr> <chr> <dbl>   <dbl>    <dbl>
1 001   bl       28      28        0
2 001   m06      27      28       -1
3 001   m12      25      28       -3
4 002   bl       30      30        0
5 002   m06      29      30       -1
6 002   m12      28      30       -2
```

Discussion

How it works: `expr()` captures an expression without evaluating it, while `!!` inside `expr()` forces evaluation of specific parts. The result is a list of named expressions. The `!!!` (splice) operator unpacks this list into `mutate()`, equivalent to writing each expression as a separate argument.

Advantages:

- Build expressions programmatically before evaluation
- Useful when the number of columns is dynamic
- Expressions can be inspected, modified, or logged before use
- Clear separation between expression construction and evaluation

Disadvantages:

- More verbose for simple cases
- Requires understanding quasiquotation deeply
- Overkill for straightforward dynamic naming

When to use: Complex metaprogramming where expressions must be built dynamically, such as generating an unknown number of columns based on input data, or when inspecting or logging expressions before evaluation is desirable.

Figure 2: A brass stencil set with interchangeable letter plates beside a partially stenciled label: programmatically generating a name.

Approach 6: data.table

The data.table package has its own `:=` operator that predates tidyverse adoption:

```
library(data.table)

calculate_change_dt <- function(data, measure) {
  bl_col <- paste0(measure, "_bl")
  cng_col <- paste0(measure, "_cng")

  dt <- as.data.table(data)

  dt[, (bl_col) := .SD[[measure]][vis == "bl"][1], by = rid]
  dt[, (cng_col) := .SD[[measure]] - .SD[[bl_col]], by = rid]

  dt[]
}

calculate_change_dt(df, "mmse")
```

```
      rid    vis  mmse mmse_bl mmse_cng
   <char> <char> <num>   <num>    <num>
1:    001     bl    28      28        0
2:    001    m06    27      28       -1
3:    001    m12    25      28       -3
4:    002     bl    30      30        0
5:    002    m06    29      30       -1
6:    002    m12    28      30       -2
```

Discussion

How it works: In data.table, wrapping a variable in parentheses on the left-hand side of `:=` forces evaluation: `(bl_col)` evaluates to `"mmse_bl"`. Without parentheses, `bl_col := value` would create a column literally named "bl_col". The `.SD` pronoun (Subset of Data) references the current group's data.

Advantages:

- Extremely fast, especially for large datasets
- Memory efficient (modifies in place by reference)
- Mature, battle-tested codebase
- The `by` argument handles grouping concisely

Disadvantages:

- Different mental model from base R and tidyverse
- Modify-by-reference semantics can cause subtle bugs
- Less readable for those unfamiliar with data.table idioms

- Parentheses syntax `(col)` is easy to forget

When to use: Performance-critical applications, very large datasets (millions of rows), or codebases already using data.table.

Figure 3: Minimalist high-speed data tools.

Approach 7: collapse

The collapse package offers high-performance data manipulation with its own idioms:

```
library(collapse)

calculate_change_collapse <- function(data, measure) {
  bl_col <- paste0(measure, "_bl")
  cng_col <- paste0(measure, "_cng")

  bl_data <- fsubset(data, vis == "bl")
  bl_values <- get_vars(bl_data, c("rid", measure))
  names(bl_values)[2] <- bl_col
```

```r
  result <- join(data, bl_values, on = "rid", how = "left")
  result[[cng_col]] <- result[[measure]] - result[[bl_col]]

  result
}

calculate_change_collapse(df, "mmse")
```

```
left join: data[rid] 6/6 (100%) <3:1st> bl_values[rid] 2/2 (100%)
```

```
# A tibble: 6 x 5
  rid   vis    mmse mmse_bl mmse_cng
  <chr> <chr> <dbl>   <dbl>    <dbl>
1 001   bl       28      28        0
2 001   m06      27      28       -1
3 001   m12      25      28       -3
4 002   bl       30      30        0
5 002   m06      29      30       -1
6 002   m12      28      30       -2
```

Discussion

How it works: collapse provides fast versions of common operations (`fsubset`, `get_vars`, `join`, etc.). Unlike dplyr, collapse does not use tidy evaluation, so dynamic column access requires base R syntax like `get_vars()` with character vectors and direct `[[` assignment. The approach above uses a join strategy rather than grouped mutation.

Advantages:

- Extremely fast (often faster than data.table for certain operations)
- Low memory footprint
- Works with both data.frames and data.tables
- Good for time series and panel data

Disadvantages:

- Smaller user community than tidyverse or data.table
- Inconsistent tidy evaluation support across functions
- Often requires mixing collapse functions with base R syntax
- Less comprehensive documentation

When to use: Performance-critical code, especially time series or panel data analysis. Also useful when speed is needed but data.table's reference semantics are to be avoided.

Comparison Summary

Approach	Dependencies	Readability	Performance	Best For
Base R `[[`	None	Moderate	Low	No-dependency scripts
Base R `do.call`	None	Low	Moderate	Package development
Classic `!! :=`	dplyr	Low	Moderate	Legacy tidyverse code
Modern glue	dplyr >= 1.0	High	Moderate	New tidyverse code
rlang `!!!` splice	rlang	Moderate	Moderate	Complex metaprogramming
data.table	data.table	Moderate	High	Large datasets
collapse	collapse	Moderate	Very High	Performance-critical work

Practical Example: Multiple Measures

The real power emerges when processing multiple measures:

```
df_multi <- tibble(
  rid = rep(c("001", "002"), each = 3),
  vis = rep(c("bl", "m06", "m12"), 2),
  mmse = c(28, 27, 25, 30, 29, 28),
  adas = c(12, 14, 18, 10, 12, 15),
  cdr = c(0.5, 0.5, 1.0, 0.5, 0.5, 0.5)
)

calculate_all_changes <- function(data, measures) {
  result <- data |> group_by(rid)

  for (measure in measures) {
    result <- result |>
      mutate(
        "{measure}_bl" := .data[[measure]][vis == "bl"][1],
        "{measure}_cng" := .data[[measure]] - .data[[paste0(measure, "_bl")]]
      )
  }

  result |> ungroup()
}

calculate_all_changes(df_multi, c("mmse", "adas", "cdr"))
```

```
# A tibble: 6 x 11
  rid   vis    mmse  adas   cdr mmse_bl mmse_cng adas_bl adas_cng cdr_bl cdr_cng
  <chr> <chr> <dbl> <dbl> <dbl>   <dbl>    <dbl>   <dbl>    <dbl>  <dbl>   <dbl>
1 001   bl       28    12   0.5      28        0      12        0    0.5     0
2 001   m06      27    14   0.5      28       -1      12        2    0.5     0
3 001   m12      25    18   1        28       -3      12        6    0.5     0.5
4 002   bl       30    10   0.5      30        0      10        0    0.5     0
5 002   m06      29    12   0.5      30       -1      10        2    0.5     0
6 002   m12      28    15   0.5      30       -2      10        5    0.5     0
```

Common Pitfalls

1. Forgetting := on the LHS (tidyverse)

```r
# Wrong: creates column literally named "{measure}_bl"
mutate("{measure}_bl" = value)

# Correct: evaluates the string
mutate("{measure}_bl" := value)
```

2. Forgetting parentheses (data.table)

```r
# Wrong: creates column named "bl_col"
dt[, bl_col := value]

# Correct: evaluates bl_col to get "mmse_bl"
dt[, (bl_col) := value]
```

3. Confusing column references

```r
# Ambiguous: is 'measure' a column or variable?
mutate("{measure}_bl" := measure[vis == "bl"][1])

# Clear: .data[[measure]] references the column
mutate("{measure}_bl" := .data[[measure]][vis == "bl"][1])
```

4. Reference semantics in data.table

```
dt <- as.data.table(df)
dt2 <- dt  # This is NOT a copy!
dt2[, new_col := 1]  # Modifies both dt and dt2

# Safe copy:
dt2 <- copy(dt)
```

What Did We Learn?

Lessons Learned

1. **`:=` enables dynamic LHS**: In both tidyverse and data.table, this operator allows the left side of an assignment to be evaluated
2. **`!!` is the unquote operator**: Forces immediate evaluation of an expression in tidy eval contexts
3. **Glue syntax is cleaner**: `"{var}_suffix"` is more readable than `!!paste0(var, "_suffix")`
4. **Parentheses matter in data.table**: `(col)` evaluates, `col` is literal
5. **`.data` removes ambiguity**: Always use `.data[[col]]` for column references in functions
6. **Choose based on context**: Readability (glue) vs performance (data.table) vs no dependencies (base R)

Related posts in this cluster

This post is part of the *R Language and Metaprogramming* series. Recommended reading order:

1. Post 62: The Pipe Equivalence Myth
2. **Post 63: Dynamic Column Names: Seven Approaches Compared** (this post)

Reproducibility

```
sessionInfo()
```

```
R version 4.6.1 (2026-06-24)
Platform: aarch64-apple-darwin25.4.0
Running under: macOS Tahoe 26.6.2

Matrix products: default
BLAS:   /opt/homebrew/Cellar/openblas/0.3.34/lib/libopenblasp-r0.3.34.dylib
LAPACK: /opt/homebrew/Cellar/r/4.6.1/lib/R/lib/libRlapack.dylib;  LAPACK version 3.12.1

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
```

```
time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] collapse_2.1.7    data.table_1.18.4 rlang_1.3.0       dplyr_1.2.1

loaded via a namespace (and not attached):
 [1] vctrs_0.7.3      cli_3.6.6        knitr_1.51       xfun_0.58
 [5] otel_0.2.0       generics_0.1.4   jsonlite_2.0.0   glue_1.8.1
 [9] htmltools_0.5.9  rmarkdown_2.31   evaluate_1.0.5   tibble_3.3.1
[13] fastmap_1.2.0    yaml_2.3.12      lifecycle_1.0.5  compiler_4.6.1
[17] Rcpp_1.1.1-1.1   pkgconfig_2.0.3  digest_0.6.39    R6_2.6.1
[21] tidyselect_1.2.1 utf8_1.2.6       pillar_1.11.1    parallel_4.6.1
[25] magrittr_2.0.5   tools_4.6.1
```

Let's Connect

Have questions, suggestions, or spot an error? Let me know.

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** Contact form

I would enjoy hearing from readers who:

- Spot an error or a better approach to any of the code in this post.
- Have suggestions for topics to cover.
- Want to discuss R programming, data science, or reproducible research.
- Have questions about anything in this tutorial.
- Simply want to say hello and connect.