# Runnable and Readable Code

## 4 weird tricks AER doesn't want you to know

Miriam Gold

August 2025

The University of Chicago

# Roadmap

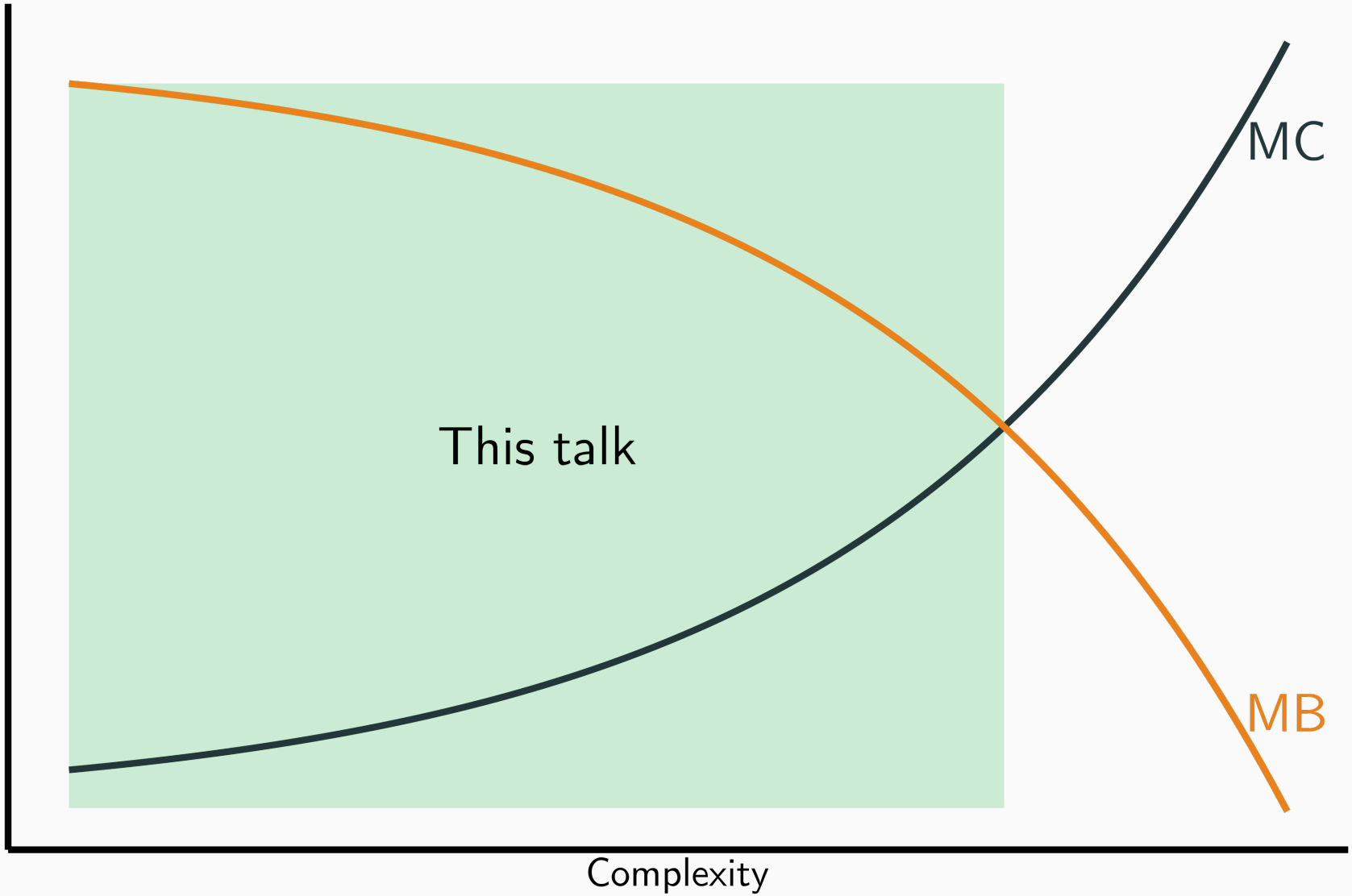
What does your successor need?

Write this down

Getting into the weeds

# What can you control?

...and what is worth controlling?



# What does your successor need?

**What they would appreciate**

- Code that runs out of the box $\rightarrow$ Runnable
- Code that they can understand $\rightarrow$ Readable

**What they (generally) don't need**

- Code that is performance optimized within an inch of its life
- Code so expertly modularized that you need track down eight nested scripts/functions just to understand one operation

# Roadmap

What does your successor need?

Write this down

Getting into the weeds

# If nothing else, do this

1. Start each script with a description of *what* it contains and *why* it exists
2. Load inputs before doing anything else, save outputs after doing everything else
3. When you are finished writing a script, clear your environment and rerun everything
4. Use master scripts

# Roadmap

What does your successor need?

Write this down

Getting into the weeds

# If nothing else, do this

1. Start each script with a description of *what* it contains and *why* it exists
2. Load inputs before doing anything else, save outputs after doing everything else
3. When you are finished writing a script, clear your environment and rerun everything
4. Use master scripts

# Preambles

Within 30 seconds, a reader should know

- Reason for existence
- Inputs
- Outputs

```
# ------------------------------------------------------------------------------ #
#'
#' Description: Determine whether DHS clusters are within 150km of a "dirty"
#'              LASSO destination grid cell
#' Author: Miriam Gold
#' Date: 5 June 2024
#' Last revised: date, initial
#' Notes: By "dirty," I mean a LASSO destination grid cell that contains origin
#'        BA pixels. We care about this because we want to test whether LASSO
#'        performance is significantly better in areas where the origin and
#'        destination overlap
#'
# ------------------------------------------------------------------------------ #
```

> We care about this because we want to test whether LASSO performance is significantly better in areas where the origin and destination overlap

# If nothing else, do this

1. Start each script with a description of *what* it contains and *why* it exists
2. Load inputs before doing anything else, save outputs after doing everything else
3. When you are finished writing a script, clear your environment and rerun everything
4. Use master scripts

# I/O

- Many scripts look like this

$$\texttt{inputs} \rightarrow \texttt{operations} \rightarrow \texttt{inputs} \rightarrow \texttt{operations} \rightarrow \texttt{outputs} \rightarrow \texttt{inputs} \rightarrow \texttt{operations} \rightarrow \texttt{outputs} \rightarrow \texttt{operations}$$

- This mimics thought, not reading
  - Difficult to skim
  - Often long
  - Error prone
- Use i/o to define the borders of a task. Think functionally

$$\texttt{inputs} \rightarrow \texttt{operations} \rightarrow \texttt{outputs}$$

▸ Hot take

# If nothing else, do this

1. Start each script with a description of *what* it contains and *why* it exists
2. Load inputs before doing anything else, save outputs after doing everything else
3. When you are finished writing a script, clear your environment and rerun everything
4. Use master scripts

# Rerunning in clean environment

- Broken references are common
- Famous last words: "I know my script can run"
- IDE-specific methods
  - **RStudio** `rm(list = ls())` and `rstudioapi::restartSession()`
  - **Stata** `clear all`
  - **Jupyter** `reset -f`

# If nothing else, do this

1. Start each script with a description of *what* it contains and *why* it exists
2. Load inputs before doing anything else, save outputs after doing everything else
3. When you are finished writing a script, clear your environment and rerun everything
4. Use master scripts

# Master scripts

- 3 jobs
  1. Describe pipeline
  2. Set up environment
  3. Run the pipeline
- Benefits
  - Self-contained
  - Single-click execution of the entire pipeline
  - Forces deprecation

▸ Deprecation

```
# ------------------------------------------------------------------------------ #
#'
#' Description: description
#' Author: author
#' Date: date
#' Last revised: date, initials
#' Notes: notes
#'
# ------------------------------------------------------------------------------ #

# Set up ========================================

rm(list = ls())
SOURCE_SCRIPTS <- FALSE

## Load packages ====

## File system paths ====
path <- "root-path"

## Load font ====

## Source all custom helper functions ====

# Source individual scripts ======================

if (SOURCE_SCRIPTS) {

  ## script-description ===============
  project-path %>%
    file.path("script-name.R") %>%
    source()

}
```

# If nothing else, do this

1. Start each script with a description of *what* it contains and *why* it exists
2. Load inputs before doing anything else, save outputs after doing everything else
3. When you are finished writing a script, clear your environment and rerun everything
4. Use master scripts

# Good luck!

# Aside: don't worry about script length

- Lots of advice to keep scripts under 200, 300 or 500 lines
- But brevity is an *outcome*, not a direct cause, of good structure.
- If you stop yourself from ever reading after writing, your scripts will be reasonable lengths.
- If your scripts are still >1000 lines, you are probably either
  - not using deprecation
  - copying and pasting

return

# Aside: deprecation

Deprecation reduces the pain of project bloat. Find a style that works for you. Here's mine.

- Code level
  - While working, comment out and tag bits that aren't relevant (e.g., `@Deprecated`)
  - Once script is stable, delete the code
- Script level
  - While working, prefix unused scripts (I use `ZZZ_` for sorting)
  - Once project is stable, move `ZZZ_` scripts to `deprecated/` directory

**For git users**

- Deprecation and deletion should happen across, not within, commits. "Buffer" commits helps with forensics.

return