INTERVIEW PREPARATION · CLINICAL R PROGRAMMING

R for Clinical Statistical Programming

End-to-end SDTM, ADaM and TLF in R · package development · validation in a GxP environment · Shiny · regulatory submission. Written for a senior SAS programmer and a beginner R programmer at the same time.

Prepared for	Bhanoji Duppada
Target roles	Principal / Associate Director / Director — R Statistical Programming
Questions	192 worked interview questions and answers
Companion	20-track audiobook, same material, spoken
Built	02 August 2026

How to use this pack

You are a senior SAS programmer with more than a decade of clinical experience. You know CDISC, you have built SDTM and ADaM, produced TLFs, and shipped submission deliverables. None of that is in question here. What these roles test is whether you can do the same work in R — and, more sharply, whether you understand **WHAT CHANGES WHEN YOU DO**.

So every answer in this pack is written twice. Once from zero, for the part of you that is a beginner in R. Once as a bridge from the SAS idea you already hold, marking the places where that idea helps you and the places where it quietly misleads you. **THE BRIDGE IS THE IMPORTANT HALF.** Most people who move from SAS to R do not fail because R is hard. They fail because they write SAS in R, and it works just well enough to hide the problem.

The four blocks in every question

Block	What it is for
Short answer	What you say first, in the room. Two to four sentences.
For the SAS programmer	The bridge from what you already know, and where the analogy breaks.
Detail	The fuller explanation the follow-up question will demand.
Interview trap	The specific thing that catches people, or the question coming next.

Read the Short answer and the Trap for everything. Read the Detail for the modules you are weakest in. The code blocks are there to be typed, not read — you cannot learn syntax by looking at it.

The companion audiobook

Every module has spoken tracks covering the same ground, at **HTTPS://CLINCODER.CLOUD/R-PREP/**. The audio carries the reasoning and the traps; the PDF carries the code. Listen on a walk, daily. Recall under pressure comes from repetition, not coverage.

What these roles actually ask for

This pack is not weighted by what is fashionable in the R community. It is weighted against **FIVE LIVE JOB POSTINGS**, all fetched and confirmed open on **2 August 2026** through the employers' own applicant-tracking APIs. Every aggregator link found by ordinary web search was dead — closed reqs, 404s, or a shut-down job board — so the list below is the set that survived verification.

#	Company	Title	Level	Location	Stated band
P1	ClinChoice	Principal Data Scientist Consultant (R and CDISC)	Principal	Remote, US	not stated
P2	IQVIA	Principal R Programmer	Principal	Durham, NC	$98,200 – $273,200
P3	Novartis	Senior Principal Statistical Programmer (Neuroscience)	Snr Principal	London, hybrid	£49,140 – £91,260
P4	Novartis	Associate Director, Statistical Programming	Assoc Director	East Hanover, NJ	$145,600 – $270,400
P5	GSK / ViiV	Principal Statistical Programmer (HIV)	Principal	Durham, NC	not stated

What recurs, ranked

ALL FIVE POSTINGS. R as the primary production language for clinical deliverables — not "R exposure", but R writing SDTM, ADaM, TLF or analysis output. Validation and QC as an **owned responsibility**. Regulatory submission context.

FOUR OF FIVE. CDISC SDTM/ADaM by name · **Git** specifically (not generic "version control") · reproducible workflows as a stated engineering property · **R Shiny** · **tidyverse** by name · standardization, automation and reusable programs across studies.

THREE OF FIVE. Technical leadership and mentoring · **AI-enabled tooling as an explicit requirement** — see below.

TWO OF FIVE. Named submission artifacts, and they do not overlap: Define.xml and eCTD (P3), ADRG, cSDRG and TMF (P5). R Markdown / Quarto.

ONE OF FIVE. pharmaverse — naming **admiral** and **tidyCDISC**, and only as preferred, not required. GxP validation named as such.

Three findings worth changing your preparation over

① SAS IS DEMOTED IN EVERY SINGLE ONE OF THESE REQS. P1 asks only for "working knowledge". P2 says SAS is "a plus, but not required". P3 lists it as one of R/SAS/Python. P5 does not mention it at all, asking

instead for "R, Python, and open-source analytics tools within regulated clinical development environments". Your SAS depth is the credibility that gets you in the room — it is not what the job is. Lead with what you can do in R, and use SAS as the evidence that you understand the deliverable.

② THE SENIORITY SIGNAL IS OWNERSHIP OF THE QC FRAMEWORK, NOT EXECUTION OF QC. P1 and P2 ask you to perform QC checks. P3, P4 and P5 make you accountable for validation, audit readiness, inspection readiness and governance. At Principal and above, "how do you validate a derivation" is the junior version of the question. The real one is "how do you set up a framework so that a hundred derivations across a portfolio are validated consistently, and prove it to an inspector."

③ THREE OF FIVE NOW ASK ABOUT AI-ASSISTED DEVELOPMENT — AND GSK NAMES THE TOOLS. Novartis says "AI-enabled tools" in both reqs. GSK/ViiV asks for experience with **generative AI coding assistants (GitHub Copilot, Claude, Codex)** alongside Agile delivery. This is new, it is not in any standard interview-prep material, and almost no candidate will have a considered answer about using an LLM assistant inside a GxP environment. **Module 5 exists because of this finding.**

> **WHAT THE POSTINGS DO NOT ASK FOR — STATED PLAINLY, SO YOU CAN SCOPE YOUR EFFORT.** Across all five live reqs, none named R package authorship, renv, testthat, shinytest2, riskmetric, the R Validation Hub, 21 CFR Part 11, rtables, tplyr, xportr, metacore, sdtm.oak, teal, Docker, or CI/CD.
>
> Module 3 covers package development and formal validation anyway, for two reasons: you asked for it, and it is a genuine **DIFFERENTIATOR** — something to volunteer that separates you from other candidates. But treat it that way. Do not spend your last week on riskmetric when four of five postings are really asking whether you can build CDISC deliverables in R, own the validation framework, and drive Git-based standardization across a portfolio.

One currency check that will date you if you miss it

`{RISKMETRIC}` IS NO LONGER THE R VALIDATION HUB'S FLAGSHIP. Its documentation site now carries an explicit **"Maintenance Only"** lifecycle badge and directs feature requests and design discussion to a successor, `{val.meter}` — described as "a next generation {riskmetric}" — with `val.criterion` and `val.report` alongside it. Bug reports are still accepted, riskmetric remains on CRAN at 0.2.7, and it remains the foundation of the Hub's Regulatory Repository initiative, so naming it is still correct.

But a candidate who says "we'd use riskmetric" without knowing it has been superseded sounds like they read a 2022 blog post. The sentence to say out loud:

> "riskmetric has moved to maintenance-only and the Hub has shifted development to val.meter, with val.criterion and val.report alongside it. The risk-based framework is unchanged — the tooling is being rebuilt."

Full treatment in **MODULE 3, Q33** and audio track m3t4.

COVERAGE LIMITS OF THIS RESEARCH. LinkedIn, Indeed, Glassdoor and BuiltIn cannot be fetched from our environment, so any posting living only there is invisible here. The specialist R-first shops (Atorus, Veramed, Phastar, Appsilon, Cytel, Certara) were not reachable through their ATS APIs — their presence or absence could not be confirmed either way.

The five things that will decide the interview

Everything in this pack matters, but these five come up in almost every conversation at this level, and they are where candidates most often lose the room.

1. "IS R VALIDATED?" — It is a badly formed question and the way you handle it signals your seniority instantly. The answer separates three different things: validating software, qualifying an environment, and verifying an output. Module 3 covers this in full. If you learn one thing from this pack, learn this.

2. THE MANY-TO-MANY JOIN. A SAS `MERGE` on a many-to-many BY group produces silently wrong output. dplyr warns you. Being able to explain why that warning is a feature — and what you do about it — demonstrates that you understand both languages rather than transliterating between them.

3. MISSING VALUES. `NA` is not `.`, an empty string is not missing, `NA` propagates through comparisons instead of sorting low, and `mean()` does not drop it for you. This is where a SAS programmer's instincts are most actively wrong, and it is easy to test.

4. WHAT ADMIRAL ACTUALLY IS. Not a macro library with different syntax. A set of composable derivation functions built around traceability, where you assemble the derivation rather than call a black box. Module 2.

5. SHINY REACTIVITY. If the role names Shiny, you will be asked why an app is not a script that runs top to bottom. Understand the reactive graph and modules; those two carry most of the senior-level Shiny questions. Module 4.

An honesty rule, stated once

Everything in this pack is designed to make you **ACCURATE**, not to make you sound impressive.

Where you have not done something, say so, then say what you would do. At Principal and Director level a candidate who marks the boundary of their own experience reads as senior; one who bluffs reads as a risk. That instinct will protect you better than any single answer here.

This applies with particular force to two claims. Do not claim you have shipped an R-based regulatory submission unless you have. And do not describe any tool as validated software — say what is actually true about its qualification, testing and documentation.

> **ON ACCURACY OF PACKAGE DETAILS.** R package APIs move faster than SAS procedures do. Function names and arguments in this pack were correct when written, but before you quote a specific argument in an interview, check the current package reference. Saying "the function that does X — let me check the exact argument" is a perfectly senior answer. Quoting a signature that changed two releases ago is not.

Module 1 — R Language Foundations and Data Manipulation for Clinical Programming

This module covers the R you need to answer confidently in a clinical programming interview: how R stores data, how it handles missing values, how the tidyverse and data.table actually work, and where a SAS mental model will quietly betray you. Every answer is written twice over — once for someone who has never opened R, and once as a bridge from DATA steps, PROC SQL and the macro facility. All code was checked against R 4.5.2 with dplyr 1.2.1, data.table 1.18.4, haven 2.5.5 and tidyr 1.3.2; where behaviour depends on a package version or on your company's setup, it is flagged rather than guessed.

Section 1 — Vectors, Atomic Types and Coercion

Q1. In R, what is a vector, and why do people say “everything is a vector”?

SHORT ANSWER. A vector is an ordered collection of values that all share one type — that is R’s fundamental unit of data, not the scalar. There is no scalar type in R: the number 5 is really a numeric vector of length one. This is why arithmetic, comparisons and most functions operate on whole columns at once without any loop.

FOR THE SAS PROGRAMMER. In a DATA step you think one observation at a time: the PDV holds a single value of AGE and you loop implicitly over rows. In R the whole AGE column arrives as one object and you operate on all of it in one expression. The closest SAS analogue is an array processed with a DO loop, except in R the loop is inside the compiled code and you never write it. The analogy breaks when you try to write row-by-row logic — that instinct is the single biggest source of slow, ugly R written by SAS programmers.

DETAIL. The six atomic vector types are logical, integer, double, character, complex and raw; in clinical work you use the first four and never the last two. A vector has a type (`typeof()`), a length (`length()`) and optionally attributes such as names, dim or class. Adding a `dim` attribute makes a matrix; adding a `class` and `levels` makes a factor; a list of equal-length vectors with a `row.names` attribute and class `data.frame` is a data frame. So a data frame is not a special primitive — it is a list of vectors wearing a class label.

```
age <- c(45L, 60L, 38L)          # integer vector, three subjects
typeof(age);  length(age);  is.vector(age)

# vectorised arithmetic: no loop, no PDV
age_months <- age * 12

# a "scalar" is just a length-one vector
length(5)                        # 1

# a data.frame is a list of vectors underneath
adsl <- data.frame(USUBJID = c("01-001", "01-002", "01-003"),
                   AGE     = age,
                   SEX     = c("M", "F", "M"))
typeof(adsl)                     # "list"
class(adsl)                      # "data.frame"
```

INTERVIEW TRAP. The follow-up is “so what is a list, then?” A list is also a vector — a generic vector — whose elements can be any type and any length, including other lists. So `is.vector(list(1, "a"))` is TRUE. When an interviewer says “everything is a vector”, the precise version is: R has atomic vectors (one type throughout) and generic vectors (lists), and almost every other structure is one of those two with attributes bolted on.

Q2. What are R’s coercion rules, and where do they bite in clinical data?

SHORT ANSWER. An atomic vector can hold only one type, so when you combine types R silently promotes everything to the most permissive one. The hierarchy is logical, then integer, then double, then character. Character wins every time, which is why one stray text value in a numeric column turns the whole column into text.

FOR THE SAS PROGRAMMER. SAS has exactly two types, numeric and character, and it refuses to mix them — you get a NOTE about character values being converted to numeric, or an error. R has more types and coerces silently, with no note in the log. That silence is the danger: SAS shouts at you, R does not. There is no `INPUT()` / `PUT()` ceremony in R, but there is also no warning that a conversion happened.

DETAIL. Coercion happens whenever you use `c()`, whenever you assign into an existing vector, and whenever a function needs a common type. Explicit conversion uses `as.numeric()`, `as.integer()`, `as.character()`, `as.logical()`. `as.numeric()` on unconvertible text gives `NA` with a warning — "NAs introduced by coercion" — and that warning is the one you must never ignore in a validated program. A second trap is that logicals coerce to 0/1 under arithmetic, which is genuinely useful: `sum(ae$AESER == "Y")` counts serious events because TRUE becomes 1.

```
c(TRUE, 1L, 2.5)                 # 1.0 1.0 2.5   -> double
c(1, "2")                        # "1" "2"       -> character

# the classic: a lab result column with "<0.5" in it
lb_char <- c("1.2", "3.4", "<0.5", "2.0")
as.numeric(lb_char)              # 1.2 3.4 NA 2.0  + warning

# logical arithmetic is a feature, not an accident
ae <- data.frame(AESER = c("Y", "N", "Y", "N"))
sum(ae$AESER == "Y")             # 2
mean(ae$AESER == "Y")            # 0.5  -> proportion for free

# integer vs double matters for equality
0.1 + 0.2 == 0.3                 # FALSE
isTRUE(all.equal(0.1 + 0.2, 0.3)) # TRUE
```

INTERVIEW TRAP. "Why did `0.1 + 0.2 == 0.3` return FALSE?" This is floating-point representation, identical to SAS and every other language — but SAS programmers rarely hit it because they compare with rounded values or fuzz. In R use `all.equal()` (returns TRUE or a descriptive string, so wrap it in `isTRUE()`) or `dplyr::near()`. Never use `==` on doubles in a derivation that decides a flag.

Q3. What is recycling, and when is it dangerous?

SHORT ANSWER. When two vectors of different lengths meet in an operation, R repeats the shorter one until it matches the longer. If the longer length is an exact multiple of the shorter, R does it silently; otherwise it warns. It is convenient for length-one values and hazardous for everything else.

FOR THE SAS PROGRAMMER. There is no direct SAS equivalent, because a DATA step statement operates on one PDV value at a time — `x = y * 2` cannot mismatch lengths. The closest cousin is a RETAIN or an array index running past its bound, which SAS treats as an error. R treats a mismatch as a feature.

DETAIL. Recycling is why `age * 12` works: `12` is length one and is recycled across every element. That case is intentional and safe. The danger is recycling a length-3 vector into a length-100 column, which silently repeats a pattern and produces wrong data with no error. Modern tidyverse code protects you here: `dplyr::mutate()` only allows a result of length 1 or length n, and errors otherwise — one of the strongest arguments for tidyverse over base R in a regulated environment.

```
c(1, 2, 3, 4) + c(10, 20)         # 11 22 13 24  (silent, exact multiple)
c(1, 2, 3)    + c(10, 20)         # 11 22 13  + warning: longer object length
                                  #   is not a multiple of shorter object length

adsl <- data.frame(USUBJID = sprintf("01-%03d", 1:4))
adsl$ARM <- c("Placebo", "Active")     # base R: silently recycles!  4 rows filled

library(dplyr)
adsl |> mutate(ARM = c("Placebo", "Active"))
# Error: `ARM` must be size 4 or 1, not 2   <- dplyr refuses
```

INTERVIEW TRAP. The follow-up is “so is recycling ever what you want?” Yes — length-one recycling, which is how you set a constant column (`mutate(STUDYID = "ABC-101")`). The interview answer is: length-one recycling is intentional, anything else should be treated as a bug, and you should prefer tools that error rather than warn.

Section 2 — data.frame vs tibble vs data.table

Q4. What is the difference between a data.frame, a tibble and a data.table?

SHORT ANSWER. All three represent a rectangular dataset, and a tibble and a data.table are both data.frames — they inherit the class, so anything expecting a data.frame will accept them. They differ in printing, in how subsetting behaves, and most importantly in whether modifying them copies the data. tibble is the tidyverse default and is strict and predictable; data.table is the performance option and modifies in place.

FOR THE SAS PROGRAMMER. All three are your SAS dataset. The difference has no SAS analogue because SAS has exactly one table type. The mental shift is that in R a dataset is an in-memory object with a class, and the class changes the behaviour of the exact same syntax — `df[, "AGE"]` returns a vector from a data.frame but a one-column table from a tibble.

DETAIL. Practical differences you should be able to name:

	data.frame	tibble	data.table
`df[, "AGE"]`	drops to a vector	stays a one-column tibble	stays a one-column data.table
`df$AG` (partial name)	partial-matches to `AGE`	warns, returns NULL	returns NULL
Printing 500k rows	prints all of it	prints 10 rows plus types	prints head and tail
Strings on creation	character since R 4.0.0	always character	always character
Modify a column	copies the object	copies the object	can modify by reference with `:=`
Row names	supported	dropped, always	not used

```
library(tibble); library(data.table)
adsl_df <- data.frame(USUBJID = c("01-001", "01-002"), AGE = c(45L, 60L))
adsl_tb <- as_tibble(adsl_df)
adsl_dt <- as.data.table(adsl_df)

class(adsl_df[, "AGE"])          # "integer"   <- dropped to a vector
class(adsl_tb[, "AGE"])          # "tbl_df"    <- still a table
class(adsl_dt[, "AGE"])          # "data.table"
class(adsl_dt[, AGE])            # "integer"   <- unquoted j evaluates to a vector

adsl_df$AG                       # 45 60   <- partial matching, silently wrong-ish
adsl_tb$AG                       # NULL    + "Unknown or uninitialised column: `AG`"

inherits(adsl_tb, "data.frame")  # TRUE
inherits(adsl_dt, "data.frame")  # TRUE
```

INTERVIEW TRAP. "Which one would you use on a study?" The answer that lands: whatever the project standard already is, because consistency beats preference in a validated environment — then give a real technical reason, e.g. tibble for readability and strictness in ADaM derivations, data.table when a dataset is large enough that copying becomes the bottleneck. Never answer with a personal preference and no reason.

Q5. What does `stringsAsFactors` mean and why does its history matter?

SHORT ANSWER. Before R 4.0.0, `data.frame()` and `read.csv()` converted every character column to a factor by default. Since R 4.0.0 the default is FALSE and strings stay strings. It matters because a lot of legacy clinical code, and a lot of advice you will find online, assumes the old behaviour.

FOR THE SAS PROGRAMMER. A factor is roughly a variable with a format applied and its values stored as the underlying codes — a category with a fixed set of allowed levels. Under the old default, R was silently applying a format to every character variable you read in. Imagine SAS auto-formatting every character column on import, with the format derived from whatever values happened to be in that file.

DETAIL. The old default caused a specific class of bug: two datasets read separately got factors whose levels came from their own data, so the integer codes underneath meant different things. Combining or comparing them produced nonsense. The fix in modern R is that it simply does not happen by default, and `readr::read_csv()` / `haven::read_sas()` never did it. The residual risk is code that explicitly sets `stringsAsFactors = TRUE`, or old scripts that assume factor behaviour (like `levels()` returning something).

```
# modern R (4.0.0+)
df <- data.frame(SEX = c("M", "F", "M"))
class(df$SEX)                    # "character"

# force the old behaviour to see the hazard
old <- data.frame(SEX = c("M", "F", "M"), stringsAsFactors = TRUE)
class(old$SEX)                   # "factor"
as.integer(old$SEX)              # 2 1 2   <- codes, not data

# the historic bug: same labels, different codes
a <- factor(c("M", "F"))         # levels F, M
b <- factor(c("M"))              # levels M
as.integer(a)[1]                 # 2
as.integer(b)[1]                 # 1   <- same "M", different code
```

INTERVIEW TRAP. The follow-up is "so should I just never use factors?" No — factors are the right tool when you need a controlled, ordered set of categories, which is exactly what a treatment arm or an AE severity is. The rule is: characters for identifiers and free text, factors deliberately created at the point where you need ordering or complete category coverage in a table.

Q6. When would you actually reach for data.table on a clinical project?

SHORT ANSWER. When the dataset is large enough that memory copying or grouped aggregation dominates your runtime — typically millions of rows, such as raw lab or ECG data, PK concentration data, or a pooled safety database across studies. For a 400-subject ADSL, the choice is irrelevant and readability should win.

FOR THE SAS PROGRAMMER. data.table is closest to what you already expect from SAS: modify a dataset in place, index it with a key so BY-group operations are fast, and read a flat file quickly. `setkey()` is essentially a permanent sort that R remembers, the way a SAS index or a sorted-by flag lets PROC MEANS skip a sort. The analogy holds better here than anywhere else in R.

DETAIL. Three concrete reasons. First, `:=` modifies columns by reference — no copy of the whole table, which matters when the table is several gigabytes. Second, `fread()` is dramatically faster than `read.csv()` on large delimited files and detects types sensibly. Third, grouped operations with `keyby`

are compiled and multi-threaded. The cost is syntax that is terser and less self-documenting, and reference semantics that surprise people — a function that takes a data.table and modifies it changes the caller's object, which is unlike every other R object and needs a comment.

```
library(data.table)
lb <- data.table(USUBJID = rep(sprintf("01-%03d", 1:3), each = 4),
                 PARAMCD = rep(c("ALT", "AST"), 6),
                 AVAL    = c(30, 28, 35, 31, 22, 25, 40, 38, 29, 27, 33, 30),
                 AVISITN = rep(1:2, 6))

setkey(lb, USUBJID, PARAMCD)                # persistent sort + index

lb[, CHG := AVAL - AVAL[AVISITN == 1][1],    # add a column BY REFERENCE
     by = .(USUBJID, PARAMCD)]

lb[, .(MEAN = mean(AVAL), N = .N), keyby = .(PARAMCD, AVISITN)]
```

INTERVIEW TRAP. "What is the risk of `:=`?" It modifies the object the caller passed in. If you write a function that does `dt[, X := 1]`, the caller's data.table now has column X, with no assignment visible at the call site. Use `copy()` at the top of the function if you do not intend that. This is the one place where R's usual copy-on-modify safety net is switched off, and interviewers love it.

Section 3 — Factors

Q7. What is a factor, and how does it relate to a SAS format?

SHORT ANSWER. A factor is an integer vector plus a `levels` attribute holding the labels — so the data is stored as codes 1, 2, 3 and displayed as "Placebo", "Low", "High". The order of the levels, not alphabetical order, controls how it sorts and how it appears in a table.

FOR THE SAS PROGRAMMER. It is the closest thing R has to a user-defined format, with one crucial difference: a SAS format is a separate object applied at display time, and the underlying variable keeps its own value. An R factor is the storage — the original character strings are gone unless you convert back. So a factor is more like a format that has been permanently baked into the variable. `as.character(f)` gives you the labels back; `as.integer(f)` gives you the codes.

DETAIL. You create factors deliberately with `factor(x, levels = ..., labels = ...)`. The two things factors buy you in clinical reporting are ordering (treatment columns in protocol order, not alphabetical) and completeness (`table()` on a factor shows levels with zero counts, which is exactly what a safety table needs — an AE term with no events in the placebo arm must still print a zero). `droplevels()` removes unused levels; `forcats::fct_relevel()` reorders; `forcats::fct_reorder()` orders one factor by a summary of another, which is how you sort a forest plot. Adding `ordered = TRUE` makes the levels comparable, so `sev >= "MODERATE"` works — but be aware that an ordered factor entering `lm()` or `glm()` gets polynomial rather than treatment contrasts, which is why coefficients suddenly come back named `.L` and `.Q`.

```
arm <- c("Active", "Placebo", "Active", "Active")

factor(arm)                                  # levels: Active, Placebo  (alphabetical)
trt <- factor(arm, levels = c("Placebo", "Active"))   # protocol order

as.integer(trt)                              # 2 1 2 2
as.character(trt)                            # back to the strings
levels(trt)                                  # "Placebo" "Active"

# the reason factors matter for a safety table: zero cells still appear
sev <- factor(c("MILD", "SEVERE"), levels = c("MILD", "MODERATE", "SEVERE"))
table(sev)
# sev
#     MILD MODERATE   SEVERE
#        1        0        1     <- MODERATE row survives
```

INTERVIEW TRAP. "How do you convert a factor of numbers back to numbers?" `as.numeric(f)` returns the codes, not the values — a genuinely dangerous silent bug when someone factorises a visit number. The correct idiom is `as.numeric(as.character(f))`. Being able to say that without hesitating signals you have actually been burned by it.

Q8. Where do factors wreck a merge or a sort?

SHORT ANSWER. Sorting, because a factor sorts by level order and a character sorts alphabetically, so the same data sorts two different ways depending on type. Merging, because joining a factor to a character forces a common type, and joining two factors with different level sets has to reconcile them.

FOR THE SAS PROGRAMMER. In SAS, a formatted variable still merges on its raw value, so formats never affect a MERGE. In R the factor is the value, so type mismatch is real. This is the analogy that actively misleads: "it's just a format, it can't affect the join" is true in SAS and false in R.

DETAIL. With dplyr 1.1.0 and later, joins use vctrs type rules: factor joined to character yields character, and two factors with different levels combine to a factor whose levels are the union. That is sane, but it means the output column type may not be what either input had — and if a downstream step depends on level order, it has silently changed. In base `merge()`, mixed types generally end up as character. The defensive habit in clinical code is to keep key variables (USUBJID, PARAMCD, AVISIT) as character everywhere and only create factors at the very last step, in the reporting layer.

```
library(dplyr)
adsl <- tibble(USUBJID = factor(c("01-001", "01-002")), ARM = c("A", "P"))
adae <- tibble(USUBJID = c("01-001", "01-003"),          AEDECOD = c("HEADACHE", "NAUSEA"))

j <- left_join(adae, adsl, by = "USUBJID")
class(j$USUBJID)                # "character"  <- type changed by the join

# sorting depends entirely on type
v <- c("SEVERE", "MILD", "MODERATE")
sort(v)                                                  # MILD, MODERATE, SEVERE (alpha)
sort(factor(v, levels = c("MILD", "MODERATE", "SEVERE")))  # MILD, MODERATE, SEVERE (clinical)
sort(factor(c("MILD","MODERATE","SEVERE"),
            levels = c("SEVERE","MODERATE","MILD")))       # SEVERE first
```

INTERVIEW TRAP. The follow-up is usually "how do you make sure a table's rows come out in protocol order?" The answer is a factor with explicit `levels =` created in the reporting step, not a sort on a character column and not `arrange()` on text. And you must be able to say why: because alphabetical order and clinical order agree by accident sometimes and never when it matters.

Section 4 — Missing Values: NA, NULL, NaN

Q9. Explain the difference between NA, NULL, NaN and NA_character_.

SHORT ANSWER. `NA` is a missing value that occupies a position in a vector. `NULL` is the absence of any value — it has length zero and disappears when you put it in a vector. `NaN` is "not a number", the result of an undefined numeric operation like `0/0`. `NA_character_`, `NA_integer_`, `NA_real_` are typed NAs, used when a function needs to know what kind of missing you mean.

FOR THE SAS PROGRAMMER. `NA` is the analogue of SAS's `.` for numerics — but there is no analogue of SAS's blank-string-means-missing convention, and that is the trap. In SAS, a character variable is missing when it equals `""`. In R, `""` is a perfectly valid one-character-long-zero string and is not missing; `NA_character_` is missing. A dataset imported from SAS will often have `""` where you expect NA, and `is.na()` will return FALSE for every one of them.

DETAIL. Key behaviours to state precisely. `NA` is logical by default, which is why it coerces into whatever vector it lands in. `is.na(NaN)` is TRUE but `is.nan(NA)` is FALSE — NaN counts as missing, NA is not a NaN. `NULL` in a list assignment deletes the element, which is how you drop a column from a data.frame. And crucially `NA == NA` is `NA`, not TRUE, so you can never test for missing with `==` — always `is.na()`.

```
length(NA)                        # 1  <- occupies a slot
length(NULL)                      # 0  <- occupies nothing
c(1, NA, 3)                       # 1 NA 3   (length 3)
c(1, NULL, 3)                     # 1 3      (length 2!)

is.na(NaN)                        # TRUE
is.nan(NA)                        # FALSE
NA == NA                          # NA   <- never test missingness with ==

# the SAS import trap: "" is not missing in R
aeout <- c("RECOVERED", "", NA)
is.na(aeout)                      # FALSE FALSE TRUE
sum(aeout == "", na.rm = TRUE)    # 1  <- you must handle both

# NULL deletes a list/data.frame element
adsl <- data.frame(USUBJID = "01-001", TEMP = 1)
adsl$TEMP <- NULL
names(adsl)                       # "USUBJID"
```

INTERVIEW TRAP. "How would you clean a SAS-derived dataset for this?" The expected answer is a step that converts empty strings to NA across character columns — for example `mutate(across(where(is.character), ~ na_if(.x, "")))` — and a statement that you agree the convention with the team up front, because converting silently can also break a comparison to a legacy SAS output that expects blanks.

Q10. How does NA propagate, and how does that differ from SAS missing?

SHORT ANSWER. Almost any operation involving NA returns NA, including comparisons — so `NA > 5` is NA, not FALSE. Aggregation functions return NA unless you pass `na.rm = TRUE`. SAS behaves completely differently: a missing numeric is treated as smaller than every real number, so `if x < 5` is TRUE when x is missing.

FOR THE SAS PROGRAMMER. This is the single most dangerous difference in the entire language for you, because both languages run without error and give different subject counts. In SAS, `if AGE < 65 then AGEGR = 'YOUNG';` assigns YOUNG to subjects with missing age. In R, `filter(AGE < 65)` drops those subjects, because the condition is NA and `filter()` keeps only TRUE. Same intent, different population, no warning in either language.

DETAIL. The rules worth stating: comparisons with NA give NA; `sum()`, `mean()`, `min()`, `max()`, `sd()` return NA unless `na.rm = TRUE`; `dplyr::filter()` keeps only rows where the condition is TRUE and therefore drops NA; base `[` with a logical index containing NA produces an NA row, which is worse than dropping. `%in%` is the exception that behaves usefully — it never returns NA, and `NA %in% c(NA, 2)` is TRUE because `match()` treats NA as matchable. Also note `if (NA)` is a hard error: "missing value where TRUE/FALSE needed".

```
library(dplyr)
NA > 5                            # NA
mean(c(45, NA, 60))               # NA
mean(c(45, NA, 60), na.rm = TRUE) # 52.5

adsl <- tibble(USUBJID = c("01-001","01-002","01-003"), AGE = c(45, NA, 70))

adsl |> filter(AGE < 65)                      # 1 row  <- 01-002 silently dropped
adsl |> filter(AGE < 65 | is.na(AGE))         # 2 rows <- explicit, auditable

adsl[adsl$AGE < 65, ]             # base R: gives an all-NA phantom row
which(adsl$AGE < 65)              # 1  <- which() drops NA, safer index

NA %in% c(1, 2)                   # FALSE
NA %in% c(NA, 2)                  # TRUE   <- the exception
if (NA) 1                         # Error: missing value where TRUE/FALSE needed
```

INTERVIEW TRAP. "Your subject count doesn't match the SAS program — where do you look first?" Missing values in a filter condition, every time. The follow-up they want is that you make the missing handling explicit in the code (`| is.na(AGE)`) rather than relying on the language default, so a reviewer can see the intent in the source.

Q11. What does `na.rm = TRUE` actually do, and when is it wrong?

SHORT ANSWER. It tells an aggregation function to discard NAs before computing, rather than returning NA. It is wrong whenever the denominator matters — because it silently changes what you divided by, and nobody reading the output can tell how many records were dropped.

FOR THE SAS PROGRAMMER. SAS's summary procedures do this for you by default: PROC MEANS ignores missing values and reports N alongside the mean, so the reader always sees the denominator. R's default is the opposite — it returns NA to force you to notice — and `na.rm = TRUE` throws that safety away without reporting N. That is why in R you should compute N in the same summarise call.

DETAIL. Always report the non-missing count next to any `na.rm = TRUE` statistic; that is standard practice in clinical summary tables anyway. Related edge cases: `sum(numeric(0))` is 0 (mathematically correct for an empty sum), but `max(numeric(0))` gives `-Inf` with a warning — which is exactly what happens when `na.rm = TRUE` removes everything in a group. That `-Inf` will flow into a table as a real number if you do not guard it.

```
library(dplyr)
vs <- tibble(USUBJID = c("01-001","01-001","01-002","01-002"),
             AVISITN = c(1, 2, 1, 2),
             AVAL    = c(120, NA, 130, 128))

# wrong: no denominator visible
vs |> summarise(MEAN = mean(AVAL, na.rm = TRUE), .by = AVISITN)

# right: N travels with the statistic
vs |> summarise(N    = sum(!is.na(AVAL)),
                NMISS = sum(is.na(AVAL)),
                MEAN = mean(AVAL, na.rm = TRUE),
                SD   = sd(AVAL,   na.rm = TRUE),
                .by  = AVISITN)

max(numeric(0))                  # -Inf + warning: no non-missing arguments to max
max(c(NA, NA), na.rm = TRUE)     # -Inf + warning  <- an empty group after removal
```

INTERVIEW TRAP. "What is `sd()` of a single value?" NA, because the denominator is n−1 and n−1 is zero. A group with exactly one non-missing observation gives you a mean and an NA standard deviation, which is correct but looks like a bug in the output — you need to know it is expected and format it as a blank rather than chase it.

Section 5 — Subsetting

Q12. What is the difference between `[`, `[[` and `$`?

Short answer. `[` keeps the container and returns the same type of thing you started with, possibly with several elements. `[[` reaches inside and pulls out one element, discarding the container. `$` is a convenience shorthand for `[[` by name, with partial matching on a data.frame.

For the SAS programmer. There is no equivalent, because SAS has no nested containers — you reference a variable by name inside a DATA step and that is the end of it. The mental image that helps: a data.frame is a train of carriages. `[` gives you a shorter train; `[[` gives you what is inside one carriage. `train[1]` is still a train; `train[[1]]` is the cargo.

Detail. For a data.frame, `df["AGE"]` returns a one-column data.frame, `df[["AGE"]]` and `df$AGE` return the AGE vector. For a list, `lst[1]` is a one-element list and `lst[[1]]` is the element. `[[` accepts a variable

holding a name, which `$` cannot do — `df[[var]]` works, `df$var` looks for a column literally called “var”. That single fact is what you use instead of a SAS macro variable resolving a column name. `$` also does partial matching on data.frames, so `df$AG` silently returns `df$AGE` — tibbles warn instead, which is one of the better reasons to use them.

```r adsl <- data.frame(USUBJID = c(“01-001”,“01-002”), AGE = c(45L, 60L), SEX = c(“M”,“F”))

class(adsl[“AGE”]) # “data.frame” <- container kept class(adsl[[“AGE”]]) # “integer” <- container discarded identical(adsl$AGE, adsl[[“AGE”]])# TRUE

# the one that matters: a column name held in a variable v <- “AGE” adsl[[v]] # 45 60 <- works adsl$v # NULL <- looks for a column literally named “v”

# two-index form: rows, columns adsl[adsl$AGE > 50, c("USUBJID", "AGE")] adsl[adsl$SEX == “F”, ] # note the trailing comma: all columns
```

lst <- list(a = 1:3, b = “x”) class(lst[1]) # “list” class(lst[[1]]) # “integer” ```

Interview trap. “What does `df[1]` do versus `df[1, ]`?” `df[1]` selects the first column (single-argument `[` on a data.frame indexes columns, because a data.frame is a list). `df[1, ]` selects the first row. Getting this backwards is one of the most common beginner errors and interviewers use it as a quick filter.

Q13. What is the `drop = TRUE` surprise?

SHORT ANSWER. When you subset a data.frame with `[` and the result has exactly one column, R silently “drops” the data.frame wrapper and hands you back a bare vector. That breaks any code downstream that expected a table. It is controlled by the `drop` argument, which defaults to TRUE for data.frames and FALSE for tibbles and data.tables.

FOR THE SAS PROGRAMMER. Nothing in SAS does this — a dataset with one variable is still a dataset. This is R deciding to be helpful and being helpful in a way that makes code non-deterministic in shape: the same line returns a data.frame or a vector depending on how many columns the selection happened to match.

DETAIL. The bug pattern is a function that selects columns from a vector of names. If the vector has two names you get a data.frame; if a filter upstream leaves one name you get a vector, and `nrow()` on it returns NULL, and the next line errors with something unrelated. Fix it with `drop = FALSE` in base R, or use a tibble, or use `dplyr::select()` which never drops. Note the direction: `drop = FALSE` is the safe setting.

```
adsl <- data.frame(USUBJID = c("01-001","01-002"), AGE = c(45L, 60L), SEX = c("M","F"))

dim(adsl[, c("AGE", "SEX")])      # 2 2   -> data.frame
dim(adsl[, "AGE"])                # NULL  -> dropped to a vector!
dim(adsl[, "AGE", drop = FALSE])  # 2 1   -> safe

# how it bites in a function
cols <- c("AGE")                  # a filter upstream left one name
nrow(adsl[, cols])                # NULL  -> next line fails mysteriously

library(dplyr)
dim(adsl |> select(all_of(cols))) # 2 1   -> select() never drops
```

INTERVIEW TRAP. The follow-up is “does the same thing happen with rows?” Yes for matrices — `m[1, ]` drops to a vector — but not for data.frames, where single-row selection keeps the data.frame. So the drop rule applies to columns on data.frames and to both dimensions on matrices, which is why `apply()` on a one-row selection sometimes behaves oddly.

Section 6 — dplyr Verbs

Q14. Walk me through the core dplyr verbs and their SAS equivalents.

SHORT ANSWER. `filter()` keeps rows, `select()` keeps columns, `mutate()` creates or changes columns, `arrange()` sorts, `group_by()` plus `summarise()` collapses to one row per group. They all take a data frame first and return a data frame, which is what lets you chain them with a pipe.

FOR THE SAS PROGRAMMER. `filter()` is a WHERE or subsetting IF. `select()` is KEEP/DROP. `mutate()` is an assignment statement in a DATA step. `arrange()` is PROC SORT. `group_by()` plus `summarise()` is PROC MEANS with a CLASS or a DATA step with BY and RETAIN. The analogy breaks in one important way: a DATA step does all of these in one pass over the data with a PDV, so you can mix row logic and aggregation freely. dplyr verbs are separate passes, and you cannot see the "previous row" without asking for it explicitly with `lag()`.

DETAIL. Everything in dplyr is non-destructive: the verbs return a new data frame and never modify the input. `summarise()` reduces rows; `mutate()` preserves them, so a grouped `mutate()` is how you attach a group statistic back onto every row — the direct replacement for RETAIN-based carry-forward logic. `n()` gives the group size, `row_number()` the position within the group, `first()` / `last()` / `lag()` / `lead()` reach across rows.

```
library(dplyr)
adae <- tibble(
  USUBJID = c("01-001","01-001","01-002","01-003","01-003","01-003"),
  AEDECOD = c("HEADACHE","NAUSEA","HEADACHE","RASH","HEADACHE","NAUSEA"),
  AESEV   = c("MILD","MODERATE","SEVERE","MILD","MILD","SEVERE"),
  TRTEMFL = c("Y","Y","Y","N","Y","Y"))

adae |>
  filter(TRTEMFL == "Y") |>                       # WHERE TRTEMFL = 'Y'
  select(USUBJID, AEDECOD, AESEV) |>              # KEEP
  mutate(SEVN = case_when(AESEV == "MILD"     ~ 1L,
                          AESEV == "MODERATE" ~ 2L,
                          AESEV == "SEVERE"   ~ 3L)) |>
  arrange(USUBJID, desc(SEVN)) |>                 # PROC SORT
  summarise(NAE = n(), MAXSEV = max(SEVN), .by = USUBJID)   # PROC MEANS BY

# grouped mutate = RETAIN-style carry-back onto every row
adae |> mutate(AESEQ = row_number(), NAE = n(), .by = USUBJID)
```

INTERVIEW TRAP. "What's the difference between a grouped `mutate()` and a `summarise()`?" `summarise()` collapses each group to one row; `mutate()` keeps every row and recycles the group-level value back across them. Most "how do I flag the subject's worst AE on every record" questions are a grouped `mutate()`, and reaching for `summarise()` plus a join is the answer that marks you as still thinking in PROC MEANS.

Q15. What is `.by` and why was it added?

SHORT ANSWER. `.by` is an argument added in dplyr 1.1.0 that applies grouping to a single verb only, and always returns an ungrouped result. It replaces the `group_by()` / verb / `ungroup()` sandwich, and it removes a whole class of bug where a grouping stayed attached longer than intended.

FOR THE SAS PROGRAMMER. `group_by()` is like setting a BY statement that then stays in effect for every subsequent step until you cancel it — which SAS would never do. `.by` is the sane version: the grouping applies to this one operation, like a BY on a single PROC.

DETAIL. With `group_by()`, the returned object carries the grouping in its class, and `summarise()` historically peeled off only the last grouping variable, leaving the rest — the source of the "`summarise()` has grouped output by 'X'" message that everyone ignores. If you then `mutate()` or `filter()` downstream, you are still grouped and do not realise it. `.by` sidesteps all of it. Both are supported and neither is deprecated; `group_by()` is still correct when you genuinely want a grouping to persist across several verbs. `.by` accepts tidyselect syntax: `.by = c(USUBJID, PARAMCD)`.

```
library(dplyr)
lb <- tibble(USUBJID = rep(c("01-001","01-002"), each = 3),
             PARAMCD = rep("ALT", 6),
             AVISITN = rep(1:3, 2),
             AVAL    = c(30, 45, 38, 22, 60, 55))

# old sandwich
lb |> group_by(USUBJID, PARAMCD) |>
      summarise(MAXALT = max(AVAL), .groups = "drop")

# same result, one line, guaranteed ungrouped
lb |> summarise(MAXALT = max(AVAL), .by = c(USUBJID, PARAMCD))

# the classic bug .by prevents
res <- lb |> group_by(USUBJID) |> mutate(BASE = AVAL[AVISITN == 1])
is_grouped_df(res)                # TRUE  <- still grouped, silently
nrow(res |> filter(AVAL == max(AVAL)))   # per-subject max, not overall!
```

INTERVIEW TRAP. "Why does `summarise()` print a message about grouped output?" Because with two or more `group_by()` variables it drops only the last one. Say that you set `.groups = "drop"` explicitly, or use `.by`, precisely so that the behaviour is visible in the code rather than in a console message a reviewer will never see.

Q16. Explain `across()` and when you would use it.

SHORT ANSWER. `across()` applies the same transformation to several columns inside `mutate()` or `summarise()`, chosen by name, by pattern, or by a predicate like `where(is.numeric)`. It replaces the copy-paste of writing the same expression once per column.

FOR THE SAS PROGRAMMER. This is your ARRAY plus DO loop, or a `%DO` loop in a macro that writes one statement per variable. The difference is that `across()` selects the columns at run time from the actual data, so it adapts if a column is absent — closer to `VAR _NUMERIC_` than to a hard-coded array.

DETAIL. The shape is `across(.cols, .fns, .names)`. `.cols` uses tidyselect: `starts_with("AVAL")`, `all_of(vec)`, `where(is.character)`. `.fns` is one function, a lambda written with `\(x)` or `~ .x`, or a named list of functions — a named list produces one output column per column-by-function pair, named by the `.names` glue template. Related helpers: `if_all()` and `if_any()` for filtering across many columns, and `pick()` (dplyr 1.1.0) for grabbing a set of columns as a data frame inside a verb.

```
library(dplyr)
adsl <- tibble(USUBJID = c("01-001","01-002"),
               AGE = c(45, 60), WEIGHT = c(70.4, 88.1), HEIGHT = c(1.70, 1.82),
               RACE = c("WHITE", ""), ETHNIC = c("", "HISPANIC OR LATINO"))

# clean the SAS blank-means-missing convention across all character columns
adsl |> mutate(across(where(is.character), \(x) na_if(x, "")))

# multiple statistics for multiple parameters, one call
adsl |> summarise(across(c(AGE, WEIGHT, HEIGHT),
                         list(N    = \(x) sum(!is.na(x)),
                              MEAN = \(x) mean(x, na.rm = TRUE),
                              SD   = \(x) sd(x,   na.rm = TRUE)),
                         .names = "{.col}_{.fn}"))

# keep only subjects with no missing values in any of these
adsl |> filter(if_all(c(AGE, WEIGHT, HEIGHT), \(x) !is.na(x)))
```

INTERVIEW TRAP. "How do you protect against a column that might not exist?" `all_of()` errors if a named column is missing, `any_of()` silently skips it. In a validated pipeline you almost always want `all_of()` — you want the program to fail loudly when the input dataset changed shape, not to quietly produce a table with a missing column.

Q17. `if_else` versus `ifelse` versus `case_when` — which and why?

SHORT ANSWER. `ifelse()` is base R and drops attributes, so it will destroy a Date or a factor. `dplyr::if_else()` is type-strict, keeps attributes, and has an explicit `missing =` argument for the NA case. `case_when()` handles more than two branches without nesting. In clinical code, prefer `if_else()` and `case_when()`.

FOR THE SAS PROGRAMMER. `ifelse()` / `if_else()` are a one-line IF/ELSE assignment. `case_when()` is an IF/ELSE IF/ELSE chain, or a SELECT/WHEN block, written as a vectorised expression rather than a statement. The difference from SAS is that these operate on the whole column and return a column — there is no implicit row loop, and there is no fall-through to "everything else" unless you write it.

DETAIL. `ifelse()`'s failure mode is genuinely severe: it strips the class attribute, so `ifelse(cond, some_date, NA)` returns a bare number — the days-since-1970 integer — and your ADT column becomes numeric. `if_else()` refuses to combine incompatible types at all, which turns a silent data corruption into an error at the point of the mistake. `case_when()` evaluates conditions in order and takes the first TRUE; unmatched rows get `.default` (dplyr 1.1.0+) or NA if you do not supply one. The older idiom `TRUE ~ value` as the last branch still works.

```
library(dplyr)
d <- as.Date(c("2023-05-01", "2023-06-15"))

class(ifelse(c(TRUE, FALSE), d, NA))     # "numeric"  <- Date destroyed
class(if_else(c(TRUE, FALSE), d, NA))    # "Date"     <- preserved

adsl <- tibble(USUBJID = c("01-001","01-002","01-003","01-004"),
               AGE = c(45, 70, NA, 64))

adsl |> mutate(
  AGEGR1 = case_when(AGE <  65 ~ "<65",
                     AGE >= 65 ~ ">=65",
                     .default  = "Missing"),          # NA lands here
  ITTFL  = if_else(!is.na(AGE), "Y", "N", missing = "N"))
```

INTERVIEW TRAP. "What happens to the NA row in your `case_when()`?" `AGE < 65` is NA and `AGE >= 65` is NA, so neither condition is TRUE, and the row falls through to `.default`. If you omit `.default`, it becomes NA. Interviewers ask this because in SAS the same logic would have put the missing subject in the `<65` group — so the two programs disagree, and you have to be able to say which one matches the SAP.

Q18. What does `coalesce()` do?

SHORT ANSWER. `coalesce()` returns the first non-missing value across several vectors, element by element. It is the standard way to fill a column from a fallback source, or to replace NA with a constant.

FOR THE SAS PROGRAMMER. It is exactly SAS's `COALESCE()` / `COALESCEC()` function, with one improvement: R's version does not care about type in the same split way — you use the same function name for character and numeric, but all arguments must share a common type.

DETAIL. Typical clinical use is a hierarchy of date sources — use the analysis date if present, otherwise the treatment start date, otherwise the reference date — or filling a derived flag with "N" where nothing set it. It vectorises over any number of arguments and a length-one final argument acts as the constant default.

```
library(dplyr)
ex <- tibble(USUBJID  = c("01-001","01-002","01-003"),
             EXSTDTC  = c("2023-05-01", NA, NA),
             RFSTDTC  = c(NA, "2023-06-02", NA),
             RFXSTDTC = c(NA, NA, "2023-07-10"))

ex |> mutate(TRTSDT = as.Date(coalesce(EXSTDTC, RFSTDTC, RFXSTDTC)))

# NA -> constant
tibble(SAFFL = c("Y", NA, "Y")) |> mutate(SAFFL = coalesce(SAFFL, "N"))
```

INTERVIEW TRAP. "How is that different from `if_else(is.na(x), y, x)`?" Functionally the same for two vectors, but `coalesce()` extends to any number of fallbacks without nesting, and it does not evaluate the branch logic per element. The related function they may probe next is `na_if()`, which goes the other way — it converts a specified value to NA, which is how you handle the SAS empty-string convention.

Section 7 — Joins

Q19. Compare dplyr joins with SAS MERGE and PROC SQL.

SHORT ANSWER. dplyr has `left_join()`, `inner_join()`, `right_join()`, `full_join()` for mutating joins that add columns, plus `semi_join()` and `anti_join()` for filtering joins that only keep or drop rows. They are set operations on keys, like PROC SQL, not the positional record-pairing that a DATA step MERGE performs.

FOR THE SAS PROGRAMMER. `inner_join` is `PROC SQL ... INNER JOIN` or `MERGE ... IF a AND b`. `left_join` is `MERGE ... IF a`. `full_join` is `MERGE` with no subsetting IF. `anti_join(a, b)` is `MERGE a(in=x) b(in=y); IF x AND NOT y;` and `semi_join` is the `IF x AND y` version that keeps only a's columns. Two differences bite immediately: dplyr does not require the inputs to be sorted, and dplyr never overwrites a same-named column — it renames both with suffixes, whereas a SAS MERGE silently lets the right dataset's value overwrite the left's.

DETAIL. Same-named non-key columns get `.x` and `.y` suffixes by default, controllable with `suffix =`. Keys are specified with `by = "USUBJID"`, with `by = c("USUBJID" = "SUBJID")` for differing names, or with `by = join_by(USUBJID)` which is the dplyr 1.1.0 syntax and also supports inequality joins such as `join_by(USUBJID, between(AEDT, TRTSDT, TRTEDT))`. If you omit `by`, dplyr does a natural join on all common columns and tells you which ones it used — convenient interactively, unacceptable in a validated program because a new common column upstream silently changes the join.

```
library(dplyr)
adsl <- tibble(USUBJID = c("01-001","01-002","01-003"),
               TRT01A  = c("Placebo","Active","Active"),
               SAFFL   = c("Y","Y","N"))
adae <- tibble(USUBJID = c("01-001","01-001","01-004"),
               AEDECOD = c("HEADACHE","NAUSEA","RASH"))

left_join(adae, adsl, by = "USUBJID")   # keeps 01-004 with NA treatment
inner_join(adae, adsl, by = "USUBJID")  # drops 01-004

anti_join(adae, adsl, by = "USUBJID")   # AEs with no matching subject -> a QC check
anti_join(adsl, adae, by = "USUBJID")   # subjects with no AEs

semi_join(adae, filter(adsl, SAFFL == "Y"), by = "USUBJID")  # AEs in the safety set

# explicit key naming and the modern syntax
left_join(adae, adsl, by = join_by(USUBJID))
```

INTERVIEW TRAP. "How would you check referential integrity between ADAE and ADSL?" `anti_join()` in both directions — orphan AE records, and subjects with no events — and assert that the first is zero rows. That is the answer that shows you use joins for QC, not just for assembly. The follow-up is often "what if the key is USUBJID in one and SUBJID in the other?", answered with `by = c("USUBJID" = "SUBJID")`.

Q20. What is the many-to-many trap, and how does dplyr 1.1.0 help?

SHORT ANSWER. If a key value appears multiple times on both sides of a join, every combination is produced, so a 3-by-4 match becomes 12 rows and your record count explodes. Since dplyr 1.1.0 this

raises a warning — "Detected an unexpected many-to-many relationship between `x` and `y`" — and you silence it deliberately with `relationship = "many-to-many"` once you have confirmed it is intended.

FOR THE SAS PROGRAMMER. This is where the SAS mental model is most dangerous, because SAS's two join tools disagree with each other. `PROC SQL` many-to-many produces the full Cartesian product within the BY group, exactly like dplyr. A DATA step `MERGE` does not — it pairs records positionally within the BY group and carries the last value forward, producing a quietly wrong result with a "MERGE statement has more than one data set with repeats of BY values" note in the log. So if you are porting a MERGE, dplyr's row count will not match SAS's, and dplyr is the one being honest.

DETAIL. `relationship =` accepts `"one-to-one"`, `"one-to-many"`, `"many-to-one"` and `"many-to-many"`. Setting it to anything other than many-to-many makes dplyr error if the data violates that expectation — which turns your assumption about the data into an enforced contract. That is a genuinely strong validation tool and worth volunteering in an interview. The related argument `unmatched = "error"` fails the join if any row on the required side finds no match, and `multiple =` controls which of several matches is taken.

```
library(dplyr)
ex <- tibble(USUBJID = c("01-001","01-001"), EXDOSE = c(10, 20))
ae <- tibble(USUBJID = c("01-001","01-001"), AEDECOD = c("HEADACHE","NAUSEA"))

left_join(ae, ex, by = "USUBJID")
# Warning: Detected an unexpected many-to-many relationship between `x` and `y`.
# 4 rows out of 2 -- the row count doubled

# state the intent; dplyr enforces it
left_join(ae, ex, by = "USUBJID", relationship = "many-to-many")  # no warning

adsl <- tibble(USUBJID = c("01-001","01-002"), TRT01A = c("Placebo","Active"))
left_join(ae, adsl, by = "USUBJID", relationship = "many-to-one") # asserts ADSL is unique

# and this errors, which is what you want in a validated program
try(left_join(ae, ex, by = "USUBJID", relationship = "many-to-one"))
```

INTERVIEW TRAP. "Your join returned more rows than the input — what happened?" Duplicate keys on the right-hand side. The senior answer names the diagnostic before the fix: check with `count(df, USUBJID) |> filter(n > 1)` or `anyDuplicated()`, decide whether the duplication is real data or a defect in the source, and only then choose between deduplicating, adding a key variable, or declaring `relationship = "many-to-many"`. Never silence the warning first.

Q21. How do you do a non-equi join, like matching an AE to the treatment period it fell in?

SHORT ANSWER. Use `join_by()` with an inequality or a `between()` condition. This is dplyr 1.1.0 functionality and it replaces the old pattern of joining on subject and then filtering, which materialises a huge intermediate table.

FOR THE SAS PROGRAMMER. This is a `PROC SQL` join with an inequality in the ON clause, which you have written many times. A DATA step MERGE cannot do it at all — you would MERGE on subject and then test dates in an IF, which is exactly the inefficient pattern `join_by()` removes.

DETAIL. `join_by()` supports `>=`, `<=`, `>`, `<`, `between(x, lower, upper)`, `within()` for interval overlap, and `closest()` for nearest-match. `closest()` is how you attach the nearest prior lab value or the last dose before an event — a very common ADaM derivation.

```
library(dplyr)
adsl <- tibble(USUBJID = c("01-001","01-002"),
               TRTSDT  = as.Date(c("2023-01-10","2023-02-01")),
               TRTEDT  = as.Date(c("2023-03-10","2023-04-01")))
ae   <- tibble(USUBJID = c("01-001","01-001","01-002"),
               AEDECOD = c("HEADACHE","RASH","NAUSEA"),
               ASTDT   = as.Date(c("2023-01-05","2023-02-14","2023-03-01")))

# treatment-emergent: event start within the treatment window
inner_join(ae, adsl, by = join_by(USUBJID, between(ASTDT, TRTSDT, TRTEDT)))

# nearest prior dose before each event
ex <- tibble(USUBJID = c("01-001","01-001"),
             EXSTDT  = as.Date(c("2023-01-10","2023-02-10")), EXDOSE = c(10, 20))
left_join(ae, ex, by = join_by(USUBJID, closest(ASTDT >= EXSTDT)))
```

INTERVIEW TRAP. “What happens to the subjects whose AE is outside the window?” With `inner_join()` they vanish, which is usually wrong for a TEAE flag — you normally want a `left_join()` on subject and then a derived `TRTEMFL` flag, so non-emergent events are still in the dataset marked “N”. Choosing the join type is a clinical decision, not a technical one, and saying that out loud is what separates a programmer from a coder.

Section 8 — Reshaping with tidyr

Q22. How do `pivot_longer()` and `pivot_wider()` compare to PROC TRANSPOSE?

SHORT ANSWER. `pivot_longer()` turns wide columns into rows — many columns into a name column and a value column. `pivot_wider()` does the reverse. Together they cover everything PROC TRANSPOSE does, but with explicit arguments instead of BY/ID/VAR statements, and without the `_NAME_` / `COL1` naming you then have to rename.

FOR THE SAS PROGRAMMER. `pivot_longer(cols = ...)` is `PROC TRANSPOSE VAR ...` producing `_NAME_` and `COL1`; `names_to` and `values_to` let you name those two columns directly. `pivot_wider(names_from, values_from)` is `PROC TRANSPOSE` with an `ID` statement, and the `BY` variables are simply the columns you did not mention — that is the biggest mental shift: tidyr infers the identifier columns as "everything not named", where SAS makes you list them.

DETAIL. Clinical data mostly lives long already — BDS structure, one row per subject/parameter/visit — so `pivot_wider()` is what you reach for when building a listing or a table shell with visits across the page. Key arguments: `id_cols` to state the identifiers explicitly rather than let tidyr infer them, `names_prefix`, and `values_fn` which handles the case where the key combination is not unique (without it, you get list-columns and a warning). `names_pattern` with a regex splits a compound column name into several new columns.

```
library(tidyr); library(dplyr)
vs_wide <- tibble(USUBJID = c("01-001","01-002"),
                  SYSBP_W1 = c(120, 130), SYSBP_W2 = c(118, 135),
                  DIABP_W1 = c(80, 85),   DIABP_W2 = c(78, 88))

vs_long <- vs_wide |>
  pivot_longer(cols      = -USUBJID,
               names_to  = c("PARAMCD", "AVISIT"),
               names_sep = "_",
               values_to = "AVAL")

# back to a table layout with visits across the page
vs_long |>
  pivot_wider(id_cols     = c(USUBJID, PARAMCD),
              names_from  = AVISIT,
              values_from = AVAL,
              names_prefix = "VISIT_")

# non-unique keys need values_fn or you get list-columns
vs_long |>
  pivot_wider(id_cols = USUBJID, names_from = PARAMCD,
              values_from = AVAL, values_fn = mean)
```

INTERVIEW TRAP. "What happens to types when you pivot longer?" All the value columns must land in one column, so they are coerced to a common type — pivot a numeric result and a character comment together and everything becomes character. That is a classic way to turn AVAL into text without noticing. Pivot each type separately, or use `values_transform`.

Q23. How do you make sure every treatment-arm-by-category cell appears in a summary, even with zero events?

SHORT ANSWER. Use `tidyr::complete()` to expand the data to all combinations of the grouping variables and fill the missing counts with zero, or make the grouping variables factors with all levels defined so `count()` and `table()` keep the empty levels.

FOR THE SAS PROGRAMMER. This is the `PRELOADFMT` / `COMPLETETYPES` problem, or the `SPARSE` option in PROC FREQ. In SAS you attach a format holding every category and tell the procedure to preload it. In R you either use factors with explicit levels — the direct format analogue — or expand the grid explicitly with `complete()`.

DETAIL. This is not a nicety: a safety table that omits the row for a preferred term with zero events in one arm is wrong, and it is one of the most common defects in first-attempt R TLF code. `complete()` takes the grouping columns and a `fill = list(N = 0)` argument. `tidyr::expand_grid()` or `dplyr::count()` on factors are the alternatives. Combine `complete()` with `nesting()` when some combinations are genuinely impossible and should not be created.

```
library(dplyr); library(tidyr)
adae <- tibble(TRT01A  = c("Placebo","Active","Active"),
               AEDECOD = c("HEADACHE","HEADACHE","NAUSEA"))

adae |> count(TRT01A, AEDECOD)             # 3 rows: Placebo/NAUSEA is missing

adae |>
  count(TRT01A, AEDECOD) |>
  complete(TRT01A, AEDECOD, fill = list(n = 0))     # 4 rows, the zero cell exists

# the factor route: levels drive the output, like a preloaded format
adae |>
  mutate(TRT01A  = factor(TRT01A,  levels = c("Placebo","Active")),
         AEDECOD = factor(AEDECOD, levels = c("HEADACHE","NAUSEA","RASH"))) |>
  count(TRT01A, AEDECOD, .drop = FALSE)             # 6 rows, RASH kept at zero
```

INTERVIEW TRAP. "Where does the denominator come from?" The big-N for each treatment arm comes from ADSL and the population flag, never from the number of distinct subjects in ADAE — a subject with no adverse events is in the denominator but appears in no ADAE row. Getting this wrong understates the denominator and overstates every percentage in the table, and interviewers ask it precisely because it is a clinical error dressed as a coding error.

Section 9 — data.table

Q24. Explain the data.table syntax.

SHORT ANSWER. The whole package is one idea: `DT[i, j, by]`, read as "take rows i, compute j, grouped by". `i` is the row filter, `j` is what to compute or which columns to return, `by` is the grouping. Everything else is a variation on those three slots.

FOR THE SAS PROGRAMMER. `i` is your WHERE, `j` is the assignment or the statistic, `by` is the BY statement — and unlike a DATA step, no prior sort is needed unless you want the speed of a key. It is the closest R gets to the compactness of SAS, which is exactly why people who came from SAS often end up preferring it.

DETAIL. Special symbols to know: `.N` is the number of rows in the current group, `.SD` is the subset of data for the current group (with `.SDcols` selecting which columns), `.I` is the row index, and `.GRP` the group counter. `:=` adds or modifies columns by reference, with no copy and no assignment on the left. `keyby` groups and returns the result sorted and keyed; `by` groups and returns groups in order of first appearance. Chaining is `DT[...][...]`.

```
library(data.table)
adae <- data.table(
  USUBJID = c("01-001","01-001","01-002","01-003","01-003"),
  TRT01A  = c("Active","Active","Placebo","Active","Active"),
  AEDECOD = c("HEADACHE","NAUSEA","HEADACHE","RASH","HEADACHE"),
  AESEVN  = c(1L, 2L, 3L, 1L, 2L))

adae[AESEVN >= 2]                                  # i: WHERE
adae[, .(N = .N), by = TRT01A]                     # j + by: PROC FREQ
adae[, .(NSUBJ = uniqueN(USUBJID)), keyby = .(TRT01A, AEDECOD)]   # sorted result

adae[, MAXSEV := max(AESEVN), by = USUBJID]        # add a column BY REFERENCE
adae[, `:=`(AESEQ = seq_len(.N), NAE = .N), by = USUBJID]  # several at once

adae[, lapply(.SD, max), by = USUBJID, .SDcols = c("AESEVN")]     # ARRAY-style
adae[AESEVN >= 2, .N, by = TRT01A][order(-N)]      # chaining
```

INTERVIEW TRAP. "Why is there no assignment arrow with `:=`?" Because `:=` modifies the existing table in place — there is nothing to assign. The follow-up is "what if you want a copy?" Call `copy(DT)` first. And the subtle one: `:=` inside a function silently changes the caller's object, so a function that is not meant to mutate its input must copy defensively.

Q25. When is data.table genuinely faster, and what is the cost?

SHORT ANSWER. It wins on grouped aggregation over many groups, on joins with keyed tables, on reading large delimited files with `fread()`, and above all on repeatedly modifying a large table, because `:=` avoids copying. On small clinical datasets the difference is unmeasurable and readability should decide.

FOR THE SAS PROGRAMMER. The speed argument mirrors why you use an indexed SAS dataset or a sorted BY: the tool knows where the groups are and does not have to search. `setkey()` is a permanent sort that R remembers, so subsequent joins and BY operations use a binary search instead of a scan.

DETAIL. The mechanism matters more than the benchmark. R's default copy-on-modify means that adding a column to a two-gigabyte table can transiently need four gigabytes; `:=` needs almost none. Grouped operations are implemented in C and can use multiple threads (`setDTthreads()`). The costs are real: reference semantics break the usual guarantee that a function cannot change its arguments, the syntax is dense enough to slow a reviewer down, and mixing data.table with dplyr pipelines in one program invites confusion. In a regulated setting, the reviewability cost is a genuine argument, not a stylistic one.

```
library(data.table)
big <- data.table(USUBJID = rep(sprintf("%06d", 1:50000), each = 20),
                  PARAMCD = rep(c("ALT","AST"), 500000),
                  AVAL    = rnorm(1e6, 30, 5))

setkey(big, USUBJID, PARAMCD)                       # sort once, reuse

system.time(big[, .(MEAN = mean(AVAL), N = .N), keyby = .(USUBJID, PARAMCD)])
system.time(big[, CHG := AVAL - mean(AVAL), by = .(USUBJID, PARAMCD)])  # no copy

# defensive copy inside a function
add_flag <- function(dt) {
  dt <- copy(dt)                    # without this, the caller's table changes
  dt[, HIGHFL := fifelse(AVAL > 40, "Y", "N")][]
}
```

INTERVIEW TRAP. "Have you actually measured it?" Say yes or say no honestly, and give the shape of the answer: profile with `system.time()` or `profvis`, and remember that in most clinical programs the runtime is dominated by reading the data and by a couple of joins, not by the arithmetic. Claiming a speed benefit you have never measured is a fast way to lose credibility.

Section 10 — The Pipe

Q26. What is the difference between `|>` and `%>%`?

SHORT ANSWER. `|>` is the native pipe, built into base R since 4.1.0, with no package needed. `%>%` is the magrittr pipe that the tidyverse popularised. Both pass the left-hand result as the first argument of the right-hand call. The native pipe is faster and dependency-free but stricter; magrittr's is more flexible.

FOR THE SAS PROGRAMMER. A pipe chain is the R equivalent of a sequence of DATA steps writing intermediate datasets, except no intermediate dataset is created and no name has to be invented. Where you would write `DATA b; SET a; WHERE ...; RUN;` then `DATA c; SET b; ...; RUN;`, you write one chain and never name `b`. The readability gain is the same reason you like a well-structured PROC SQL over three nested queries.

DETAIL. The differences that matter in practice: `%>%` accepts a bare function name (`x %>% sum`) while `|>` requires the call parentheses (`x |> sum()`). `%>%` uses `.` as a placeholder anywhere, including multiple times; `|>` uses `_` and only as a named argument — `d |> merge(x = _, y = other, by = "USUBJID")` works, `1:3 |> sum(_)` is an error. Since R 4.3.0, `_` also works with extraction, so `d |> _$AGE` is valid. Most teams standardise on one; the native pipe is the safer long-term bet because it has no dependency and cannot be affected by a package update.

```
library(dplyr)
adsl <- tibble(USUBJID = c("01-001","01-002","01-003"),
               AGE = c(45, 70, 64), SAFFL = c("Y","Y","N"))

# native pipe
adsl |>
  filter(SAFFL == "Y") |>
  mutate(AGEGR1 = if_else(AGE >= 65, ">=65", "<65")) |>
  count(AGEGR1)

# magrittr placeholder anywhere
library(magrittr)
adsl %>% lm(AGE ~ 1, data = .)

# native pipe: placeholder must be a NAMED argument
adsl |> lm(AGE ~ 1, data = _)        # fine
# 1:3 |> sum(_)                       # Error: pipe placeholder can only be
                                      #   used as a named argument
adsl |> _$AGE                         # works from R 4.3.0
```

INTERVIEW TRAP. "How do you debug a long pipe?" You cannot put a breakpoint in the middle of one, so the techniques are: insert a `print()` or `glimpse()` step temporarily, use `dplyr` itself (`|> {\(d) {browser(); d}}()` in a pinch), or — the honest answer most reviewers want — keep pipes short enough that each one has a name and a purpose. A twenty-step pipe is not clever, it is unreviewable, which matters more in a regulated environment than anywhere else.

Section 11 — Iteration: apply, purrr, and Vectorisation

Q27. Walk me through the apply family and the purrr equivalents.

SHORT ANSWER. `lapply()` applies a function to each element of a list or vector and always returns a list. `sapply()` does the same but tries to simplify the result, which makes its return type unpredictable. `vapply()` is `sapply()` with a required return-type template, so it is safe. `apply()` works over the rows or columns of a matrix. purrr's `map()` is `lapply()`, and `map_chr()`, `map_dbl()`, `map_int()`, `map_lgl()` are typed versions like `vapply()`.

FOR THE SAS PROGRAMMER. The closest analogue is a macro `%DO` loop that calls the same code once per dataset, per parameter or per treatment arm — a loop over things, not over rows. That is the right way to think about it: the apply family is for iterating over datasets, variables or parameters, not over observations. Iterating over observations is what vectorisation is for.

DETAIL. The reason to prefer `vapply()` or `map_dbl()` over `sapply()` is that `sapply()` returns a list when it cannot simplify, a vector when it can, and a matrix when every result has the same length greater than one — so the same line returns three different shapes depending on data. In production code that is a latent bug. `apply()` on a data.frame first coerces to a matrix, which means a mixed-type data.frame becomes all character — a nasty and very common surprise. `Map()` / `purrr::map2()` iterate over two lists in parallel, `pmap()` over any number. Note that purrr 1.0.0 superseded `map_dfr()`; the current idiom is `map()` followed by `list_rbind()`.

```
library(purrr); library(dplyr)

adsl <- data.frame(USUBJID = c("01-001","01-002"), AGE = c(45, 60))

sapply(adsl, class)                 # named character vector -- fine here
vapply(adsl, class, character(1))   # same, but the type is guaranteed

# the apply() coercion trap
apply(adsl, 1, function(r) r["AGE"])   # "45" "60"  <- character! matrix coercion

# the right shape for "run this per parameter"
params <- c("ALT", "AST", "BILI")
summaries <- map(params, \(p) {
  tibble(PARAMCD = p, N = 10L, MEAN = 30)     # your real summary code here
})
bind_rows(summaries)                 # or: summaries |> list_rbind()

# typed output, guaranteed length and type
map_chr(params, \(p) paste0("PARAM_", p))
map_int(list(1:3, 1:5), length)
```

INTERVIEW TRAP. "Why not just use `sapply()`?" Because its return type depends on the data, so a program that works in test can return the wrong shape in production when a group happens to have one element instead of three. Naming that failure mode — not just saying "vapply is safer" — is what an interviewer is listening for.

Q28. Is "avoid loops in R" good advice?

SHORT ANSWER. Partly. Vectorised operations are genuinely much faster than an element-by-element loop, so you should never loop over rows to do arithmetic. But a `for` loop over a handful of

parameters or datasets is perfectly fine, often clearer than an apply, and the performance difference is nil. The real rule is: never loop where a vectorised operation exists.

FOR THE SAS PROGRAMMER. You already have this instinct in the right place — you would not write a DO loop to add two variables. The difference is that R's slow pattern is specifically growing an object inside a loop, which has no obvious SAS parallel because SAS writes to disk sequentially. In R, appending a row to a data frame inside a loop reallocates and copies the entire object every iteration, which turns a linear job into a quadratic one.

DETAIL. The three genuine performance sins are: growing a vector or data frame inside a loop instead of pre-allocating or collecting into a list; calling a vectorised function once per element; and repeated `rbind()`. The fix for the first two is vectorisation, and for the third it is collecting results into a list and calling `bind_rows()` or `rbindlist()` once at the end. A loop that runs 12 times over 12 endpoints is not a performance problem and rewriting it as a `map()` for style reasons is not a productive use of validation effort.

```
library(dplyr)
n <- 1e5
x <- rnorm(n)

# vectorised: one call
system.time(y1 <- ifelse(x > 0, "POS", "NEG"))

# the slow pattern: growing an object
system.time({ y2 <- character(0); for (i in seq_len(n)) y2[i] <- if (x[i] > 0) "POS" else "NEG" })

# pre-allocated loop: much closer to vectorised, still slower
system.time({ y3 <- character(n); for (i in seq_len(n)) y3[i] <- if (x[i] > 0) "POS" else "NEG" })

# the correct loop pattern when you must loop: collect, then combine once
out <- vector("list", 3)
for (i in seq_along(c("ALT","AST","BILI"))) out[[i]] <- tibble(PARAMCD = c("ALT","AST","BILI")[i], N
= i)
bind_rows(out)
```

INTERVIEW TRAP. "What is wrong with `for (i in 1:length(x))`?" When `x` is empty, `length(x)` is 0 and `1:0` is the vector `c(1, 0)` — so the loop runs twice with nonsense indices instead of zero times. Use `seq_along(x)` or `seq_len(n)`, which return an empty sequence correctly. This is a classic and a fast way to demonstrate real R experience.

Section 12 — Dates and Times

Q29. What is the difference between Date and POSIXct, and when do you use each?

SHORT ANSWER. `Date` stores a calendar day as the number of days since 1970-01-01, with no time and no timezone. `POSIXct` stores an instant as the number of seconds since 1970-01-01 UTC, and carries a timezone attribute. Use `Date` for anything that is a day — visit dates, treatment start — and `POSIXct` only when the time of day genuinely matters, such as PK sampling or a dosing time.

FOR THE SAS PROGRAMMER. `Date` is a SAS date value and `POSIXct` is a SAS datetime value, but the origins differ: SAS counts from 1960-01-01, R from 1970-01-01. That ten-year offset — 3,653 days — is exactly why you must never move raw numeric date values between the two systems without conversion. `haven::read_sas()` handles the conversion for you; a CSV export of raw numeric dates does not.

DETAIL. ADaM numeric date variables (`ADT`, `ASTDT`, `TRTSDT`) map naturally to `Date`. Arithmetic on `Date` gives a `difftime` in days, so `TRTEDT - TRTSDT + 1` is the treatment duration in days — and note that `difftime` prints with a unit and needs `as.numeric()` before it goes into a numeric column. On `POSIXct`, subtraction gives a `difftime` whose units are chosen automatically (secs, mins, hours or days depending on magnitude), which is a genuine trap: the same expression can yield "3.5 hours" or "210 minutes". Always pass `units =` explicitly to `difftime()`.

```
d1 <- as.Date("2023-05-01"); d2 <- as.Date("2023-06-15")
d2 - d1                              # Time difference of 45 days  (a difftime)
as.numeric(d2 - d1) + 1              # 46  <- study day style duration

as.numeric(d1)                       # 19478  days since 1970-01-01
# the same day as a SAS date value would be 19478 + 3653 = 23131

t1 <- as.POSIXct("2023-05-01 08:30:00", tz = "UTC")
t2 <- as.POSIXct("2023-05-01 12:00:00", tz = "UTC")
t2 - t1                              # Time difference of 3.5 hours   <- units vary!
difftime(t2, t1, units = "mins")     # 210 mins -- explicit, reproducible

# ADaM study day: there is no Day 0
trtsdt <- as.Date("2023-05-01")
adt    <- as.Date(c("2023-04-28", "2023-05-01", "2023-05-10"))
diff   <- as.numeric(adt - trtsdt)
ady    <- ifelse(adt >= trtsdt, diff + 1, diff)
ady                                  # -3  1  10   <- no zero, as CDISC requires
```

INTERVIEW TRAP. "How do you compute study day?" The rule is `ADY = ADT - TRTSDT + 1` on or after treatment start, and `ADY = ADT - TRTSDT` before it — because there is no Day 0 in CDISC. Naive subtraction gives you a Day 0 that should not exist, and the interviewer is checking whether you know the CDISC convention, not the R arithmetic.

Q30. How do you handle ISO 8601 partial dates from `--DTC` variables?

SHORT ANSWER. SDTM `--DTC` variables are character, in ISO 8601, and may be partial — a year only, a year and month, or a date with no time. You cannot convert them with a plain `as.Date()` because that returns NA for anything incomplete. You parse what is present, impute the missing components according to the SAP, and record what you imputed in the corresponding `--DTF` / `--TMF` flag.

FOR THE SAS PROGRAMMER. You have written this logic many times with `SUBSTR` and `INPUT` and a chain of IF/THEN. The R version is the same logic; the difference is that the imputation flags are a hard CDISC requirement that reviewers check, and in R the pharmaverse `admiral` package implements the standard imputation rules so most teams do not hand-roll it.

DETAIL. Valid ISO 8601 partial forms include `"2023"` (year only), `"2023-05"` (year and month), `"2023-05-14"` (complete), `"2023-05-14T08:30"` (with time) and the hyphen-placeholder form `"2023---14"` where a lower component is known but a higher one is not. `as.Date()` behaves treacherously here: given a vector whose first element is complete it infers the `%Y-%m-%d` format and returns NA for the partials, but given only `"2023-05"` it throws "character string is not in a standard unambiguous format". So the same function either fails loudly or fails silently depending on the order of your data — never rely on it for `--DTC`. `lubridate::ymd()` returns NA with a warning, which is at least consistent; `lubridate::ym()` parses `"2023-05"` to the first of that month. The ADaM imputation flag `ASTDTF` takes "D" if only the day was imputed, "M" if month and day, "Y" if year, month and day — the value names the highest component imputed. `admiral::derive_vars_dt()` derives the date and the flag together with a `highest_imputation` argument, which is the route to prefer over hand-written substring logic.

```
library(lubridate); library(dplyr); library(stringr)

dtc <- c("2023-05-14", "2023-05", "2023", "2023-05-14T08:30", NA)

as.Date(dtc)                          # "2023-05-14" NA NA "2023-05-14" NA
as.Date("2023-05")                    # Error: not in a standard unambiguous format
str_length(dtc)                       # 10, 7, 4, 16, NA  <- the completeness test

tibble(AESTDTC = dtc) |>
  mutate(
    NCHAR  = str_length(AESTDTC),
    # impute to the FIRST possible day -- the SAP decides first vs last vs missing
    ASTDT  = case_when(NCHAR >= 10 ~ ymd(str_sub(AESTDTC, 1, 10)),
                       NCHAR == 7  ~ ym(AESTDTC),
                       NCHAR == 4  ~ ymd(paste0(AESTDTC, "-01-01"))),
    ASTDTF = case_when(NCHAR >= 10 ~ NA_character_,
                       NCHAR == 7  ~ "D",
                       NCHAR == 4  ~ "M"))

# in production, prefer the validated implementation:
# admiral::derive_vars_dt(dataset, new_vars_prefix = "AST",
#                         dtc = AESTDTC, highest_imputation = "M")
```

INTERVIEW TRAP. "Which way do you impute?" There is no universal answer — it is an SAP decision, and the common conservative convention is to impute an adverse-event start date to the earliest possible day (so the event is more likely to be treatment-emergent) and an end date to the latest. If you assert a single rule confidently, you are answering a statistical question as if it were a programming one. Say that it comes from the SAP and that the flag must always be populated.

Q31. What timezone traps should you know?

SHORT ANSWER. `POSIXct` carries a timezone attribute, and if you do not set one, R uses the machine's local timezone — so the same code gives different results on a laptop in London and a validated server in UTC. Set `tz = "UTC"` explicitly on every datetime you create, and never let a partial datetime string be parsed without one.

FOR THE SAS PROGRAMMER. SAS datetimes have no timezone concept at all; a datetime is just a number of seconds and you handle any offset yourself. R's automatic local-timezone behaviour therefore has no SAS analogue and catches SAS programmers repeatedly. The symptom is a date that shifts by one day between environments.

DETAIL. The dangerous case is conversion between `POSIXct` and `Date`: `as.Date(x)` on a `POSIXct` converts using UTC by default, so a datetime of 2023-05-01 23:00 in New York becomes 2023-05-02 as a Date. `lubridate::with_tz()` changes the displayed timezone while keeping the same instant; `force_tz()` keeps the displayed clock time and changes the instant. Those two are frequently confused and do opposite things. Also note `Sys.timezone()` and that daylight saving means some local clock times do not exist and some occur twice.

```
library(lubridate)
t <- as.POSIXct("2023-05-01 23:00:00", tz = "America/New_York")

as.Date(t)                              # "2023-05-02"  <- converts via UTC
as.Date(t, tz = "America/New_York")     # "2023-05-01"  <- explicit, correct

with_tz(t,  "UTC")                      # same instant, shown as 03:00 next day
force_tz(t, "UTC")                      # same clock face 23:00, a DIFFERENT instant

attr(t, "tzone")                        # "America/New_York"
Sys.timezone()                          # whatever the machine says -- do not rely on it
```

INTERVIEW TRAP. "Why did the date shift by one day in the output?" A `POSIXct` near midnight converted to `Date` without a timezone. The senior answer is preventative: if the analysis is by calendar day, store `Date`, not `POSIXct`, and do the conversion once at read time with an explicit timezone rather than letting it happen implicitly deep in a derivation.

Section 13 — Strings and Regular Expressions

Q32. Base R string functions or stringr — and what regex do you actually need?

SHORT ANSWER. Both work; stringr is a consistent wrapper over the same regex engine, with the string always the first argument, uniform naming, and sane NA handling. Base R has `grepl()`, `sub()`, `gsub()`, `substr()`, `nchar()`, `trimws()`, `sprintf()`. The regex you need day to day is small: anchors, character classes, quantifiers, and groups.

FOR THE SAS PROGRAMMER. `str_detect()` is `PRXMATCH` or `INDEX`. `str_replace_all()` is `PRXCHANGE` or `TRANWRD`. `str_sub()` is `SUBSTR`. `str_pad()` is the zero-padding you do with `PUT` and a Z format. `str_trim()` is `STRIP`. The big difference is that SAS character variables have a fixed declared length and are blank-padded; R strings are variable length with no padding, so `"AB"` and `"AB "` are genuinely different values and comparisons between them fail. That trips up everyone importing from SAS.

DETAIL. The one behavioural difference worth memorising: base `grepl("a", NA)` returns FALSE, whereas `stringr::str_detect(NA, "a")` returns NA. The stringr answer is more correct — you do not know whether a missing string matches — and the base answer silently folds missing into "no match", which changes a subject count. Regex essentials: `^` and `$` anchor, `.` is any character, `[A-Z]` a class, `\d` a digit (note the double backslash in an R string), `*` `+` `?` quantify, `()` groups and `|` alternates. Use `fixed()` when you want a literal match — matching a literal dot with `str_detect(x, ".")` matches everything.

```
library(stringr); library(dplyr)

usubj <- c("ABC-101-01-001", "ABC-101-02-014", NA)

str_detect(usubj, "^ABC-101")          # TRUE TRUE NA
grepl("^ABC-101", usubj)               # TRUE TRUE FALSE   <- NA became FALSE!

str_sub(usubj, -3, -1)                 # "001" "014" NA    <- subject number
str_replace(usubj, "^ABC-101-", "")    # strip the study prefix
str_pad("7", width = 3, pad = "0")     # "007"
str_trim("  MILD  "); str_squish("A   B")   # "MILD" ; "A B"

# extract the site from the ID with a capture group
str_match(usubj, "^([A-Z]+)-(\\d+)-(\\d+)-(\\d+)$")[, 4]   # "01" "02" NA

# the SAS trailing-blank trap
"MILD" == "MILD "                      # FALSE -- strip on import
tibble(AESEV = c("MILD ", "SEVERE")) |> mutate(AESEV = str_trim(AESEV))

# case-insensitive and literal matching
str_detect("Headache", regex("HEADACHE", ignore_case = TRUE))   # TRUE
str_detect("A.B", fixed("."))                                    # TRUE (literal dot)
```

INTERVIEW TRAP. "How would you find all AEs whose preferred term mentions cardiac terms?" The answer they want includes a caveat: you do not do medical grouping with a regex — you join to the MedDRA hierarchy on SOC or HLT, because a text search misses synonyms and catches false positives. Reaching straight for `str_detect()` on `AEDECOD` for a clinical grouping is a domain error, and a clinical interviewer will notice.

Section 14 — Reading and Writing Clinical Data

Q33. How do you read a SAS7BDAT or an XPT file into R, and write one back?

SHORT ANSWER. The `haven` package: `read_sas()` for SAS7BDAT, `read_xpt()` for XPORT transport files, and `write_xpt()` to produce one. They return tibbles and preserve SAS variable labels as an attribute on each column. For an FDA submission you must pass `version = 5` to `write_xpt()`, because the default is version 8.

FOR THE SAS PROGRAMMER. `read_xpt()` / `write_xpt()` are your `PROC COPY` to and from a transport library, or `PROC CPORT` / `CIMPORT`'s cousin. The thing that has no analogue is that once the data is in R, the label is an attribute rather than a property of the dataset definition, and R functions are free to drop it — which is exactly what happens.

DETAIL. `read_sas()` also takes a `catalog_file` argument so it can apply SAS formats stored in a format catalog. Values that carried a SAS format come back as `haven_labelled` columns, which are numeric with a `labels` attribute; `haven::as_factor()` converts them to factors using those labels, and `zap_labels()` strips them if you just want the raw values. XPORT version 5 imposes the classic constraints — variable names up to 8 characters, labels up to 40, character values up to 200 bytes — and `write_xpt()` will error rather than silently truncate, which is the behaviour you want. Confirm your own organisation's submission requirements; do not quote a rule you have not checked.

```
library(haven); library(dplyr); library(purrr)

adsl <- read_sas("adsl.sas7bdat")             # tibble, labels preserved
dm   <- read_xpt("dm.xpt")

attr(adsl$AGE, "label")                       # "Age"  <- the SAS label survives
map_chr(adsl, \(x) attr(x, "label") %||% NA_character_)   # inspect every label

# formatted (labelled) columns
as_factor(adsl$RACE)                          # apply the SAS format as levels
zap_labels(adsl)                              # strip labelled class, keep values

# writing back for submission -- version 5 is NOT the default
write_xpt(adsl, "adsl.xpt", version = 5, name = "ADSL")
```

INTERVIEW TRAP. "What is the default XPORT version and why does it matter?" `write_xpt()` defaults to version 8; regulatory submissions have historically required version 5 transport files. If you take the default you produce a file that a submission gateway may reject, and nothing in the R session tells you. Knowing that the default is not the submission format is the answer.

Q34. What is the label attribute problem and how do you handle it?

SHORT ANSWER. SAS variable labels arrive in R as an attribute on each column, and most R operations do not promise to keep attributes — a `mutate()` that recalculates a column typically drops its label. So labels erode as your program runs, and the XPT you write at the end is missing them. The fix is to apply labels deliberately at the end from a metadata specification, not to rely on carrying them through.

FOR THE SAS PROGRAMMER. In SAS the label lives in the dataset descriptor and survives every DATA step unless you overwrite it. In R it is just an attribute riding on a vector, and it survives only where the

function happens to preserve attributes. This is the single most annoying difference for anyone building submission datasets in R, and pretending otherwise in an interview will not help you.

DETAIL. The mature answer is that labels, lengths, types and formats should come from a metadata source — a define-XML-derived spec or a `metacore` object — and be applied in one controlled step at the end. The pharmaverse package `xportr` exists exactly for this: `xportr_label()`, `xportr_length()`, `xportr_type()`, `xportr_format()`, `xportr_order()` and `xportr_write()` take a spec and apply it, and they report what did not match. `labelled::set_variable_labels()` is the lighter-weight option for applying labels from a named vector or list. Treat labels as metadata applied from a spec, never as something you hand-maintain in code.

```
library(dplyr); library(haven); library(labelled)

adsl <- tibble(USUBJID = c("01-001","01-002"), AGE = c(45, 60))
var_label(adsl$AGE) <- "Age"
attr(adsl$AGE, "label")               # "Age"

adsl2 <- adsl |> mutate(AGE = AGE + 1)
attr(adsl2$AGE, "label")              # NULL  <- the label is gone

# apply labels from a spec in one controlled step
spec <- list(USUBJID = "Unique Subject Identifier", AGE = "Age")
adsl3 <- set_variable_labels(adsl2, .labels = spec)
attr(adsl3$AGE, "label")              # "Age"

# production route: apply spec-driven metadata, then write
# adsl |> xportr_type(spec) |> xportr_length(spec) |>
#         xportr_label(spec) |> xportr_write("adsl.xpt")
```

INTERVIEW TRAP. "How do you check that every variable in the output has a label?" A one-line assertion over the columns — `sapply(df, \(x) is.null(attr(x, "label")))` and fail if any are TRUE — run as part of the program, not by eyeballing the define. Interviewers are testing whether you build the check into the code or trust yourself to remember.

Q35. What do readr's read functions give you over base R's?

SHORT ANSWER. `readr::read_csv()` is faster, returns a tibble, never converts strings to factors, never mangles column names by default, and reports the column types it guessed. Base `read.csv()` guesses silently and historically converted strings to factors. The important feature is `col_types`, which lets you declare types rather than let R guess.

FOR THE SAS PROGRAMMER. `read_csv()` guessing types is like letting PROC IMPORT decide, and you already know PROC IMPORT is fine for exploration and unacceptable for production because it infers lengths and types from the first N rows. `col_types` is your INPUT statement with explicit informats — the thing that makes the read reproducible.

DETAIL. By default `read_csv()` guesses from the first 1,000 rows, so a column that is numeric for 1,000 rows and then contains "NOT DONE" produces a parsing problem flagged in `problems()`. In a validated program, always specify `col_types` — this both documents the expected structure and turns a changed source file into a loud failure. Read subject identifiers as character, always: a site number like `007` read as numeric becomes `7`, and the join to ADSL then fails.

```
library(readr)

# exploratory: let it guess, then look at what it guessed
lb <- read_csv("lb.csv")
spec(lb)
problems(lb)                          # any values that failed to parse

# production: declare the contract
lb <- read_csv("lb.csv",
               col_types = cols(USUBJID = col_character(),
                                SITEID  = col_character(),   # keep leading zeros
                                LBTESTCD = col_character(),
                                LBSTRESN = col_double(),
                                LBDTC   = col_character(),   # parse --DTC yourself
                                .default = col_character()))
```

INTERVIEW TRAP. "Why read `LBDTC` as character rather than a date?" Because `--DTC` may be partial, and any automatic date parser will turn a legitimate partial date into NA and destroy information you need for the imputation flag. Reading SDTM date-time variables as character and converting deliberately is the correct clinical habit.

Section 15 — Functions, Scoping and the Macro Question

Q36. R has no macro variables. What replaces `%LET` and `&VAR`?

SHORT ANSWER. Ordinary variables and function arguments. A SAS macro variable exists because the DATA step language cannot hold a value outside a dataset; R has no such limitation, so a "macro variable" is just a variable, and a "macro" is just a function. Where you genuinely need to pass a column name around, you use tidy evaluation — `{{ }}`, `.data[[ ]]`, or `all_of()`.

FOR THE SAS PROGRAMMER. This is the biggest conceptual adjustment. SAS macro is a text-substitution layer that runs before the compiler sees your code, which is why `&VAR` can expand into anything — a variable name, half a statement, a whole PROC. R has no separate pre-processing layer; code is evaluated directly. The upside is no macro quoting hell, no `%NRSTR`, no double-ampersand resolution. The downside is that you cannot generate arbitrary syntax by string-pasting, and you must learn a different mechanism for the specific case of "pass me a column name".

DETAIL. Three mechanisms cover almost everything. First, an ordinary argument for values: `run_summary(dataset, cutoff = 65)`. Second, `{{ }}` (the "embrace" operator from rlang, re-exported by dplyr) when a user should be able to type a bare column name. Third, `.data[[var]]` or `all_of(var)` when the column name arrives as a string, which is the case when it comes from a metadata file. For building text — a title, a file path — use `paste0()` or `glue::glue()`, which is the closest thing to `&VAR` resolution in a string.

```
library(dplyr); library(rlang)

# 1. plain arguments for plain values
count_above <- function(data, cutoff = 65) {
  data |> summarise(N = sum(AGE > cutoff, na.rm = TRUE))
}

# 2. embrace: the caller types a bare column name
summarise_param <- function(data, group_var) {
  data |> summarise(N = n(), MEAN = mean(AVAL, na.rm = TRUE), .by = {{ group_var }})
}
lb <- tibble(PARAMCD = c("ALT","ALT","AST"), AVAL = c(30, 40, 25))
summarise_param(lb, PARAMCD)

# 3. the column name arrives as a STRING, e.g. from a spec file
summarise_by_string <- function(data, group_col) {
  data |> summarise(N = n(), .by = all_of(group_col))
}
summarise_by_string(lb, "PARAMCD")
lb |> filter(.data[["PARAMCD"]] == "ALT")     # .data pronoun

# building text, the &VAR analogue
param <- "ALT"; n <- 42
glue::glue("Table 14.3.1 -- {param} (N={n})")
```

INTERVIEW TRAP. "So how do you loop a report over every parameter, the way a macro would?" A function that takes the parameter as an argument, plus `map()` or a `for` loop over the parameter list — and crucially, the function returns an object rather than writing a global. The trap is writing R that emulates macro behaviour with `assign()` and `get()` into the global environment; that works, and every reviewer will hate it.

Q37. Explain lazy evaluation and `...` in R functions.

SHORT ANSWER. R does not evaluate an argument until it is actually used, so an argument you never touch can be invalid and never cause an error. `...` collects any number of extra arguments and passes them on to another function, which is how wrappers stay flexible without listing every parameter.

FOR THE SAS PROGRAMMER. SAS macro parameters are also substituted rather than evaluated up front, so lazy evaluation will feel vaguely familiar. `...` has no SAS analogue — the closest is a macro with a long list of optional keyword parameters that it passes through to a PROC.

DETAIL. Lazy evaluation has three visible consequences. Default arguments can refer to other arguments, because the default is evaluated inside the function when needed: `function(x, n = length(x))` works. An argument that is never used is never evaluated, so `function(a, b) a` never touches `b`. And `missing(arg)` tells you whether the caller supplied a value. The classic gotcha is a loop that creates functions capturing a loop variable — by the time the function runs, the variable has moved on; `force(x)` inside the function pins it. For `...`, `list(...)` captures the values and `...length()` counts them; the risk is that a misspelled argument name falls silently into `...` and does nothing.

```
f <- function(x, y) x                # y is never used
f(1, stop("never runs"))             # 1  -- no error, y was never evaluated

g <- function(x, n = length(x)) n    # default refers to another argument
g(c(1, 2, 3))                        # 3

h <- function(x, y) { if (missing(y)) "y not supplied" else y }
h(1)                                 # "y not supplied"

# ... passes options through to an inner function
summarise_num <- function(x, ...) {
  c(N = sum(!is.na(x)), MEAN = mean(x, ...), SD = sd(x, ...))
}
summarise_num(c(45, NA, 60), na.rm = TRUE)

# the ... typo trap: a misspelt argument is silently swallowed
mean2 <- function(x, ...) mean(x, ...)
mean2(c(1, NA, 3), na.rn = TRUE)     # NA -- "na.rn" went into ... and was ignored

# the loop-capture gotcha
make <- function(i) { force(i); function() i }
fns  <- lapply(1:3, make)
fns[[1]]()                           # 1
```

INTERVIEW TRAP. "What is the danger of `...`?" It silences typos — `na.rn` instead of `na.rm` produces no error and quietly changes your result. In a validated function, either name your arguments explicitly instead of using `...`, or validate the names you received with `names(list(...))`.

Q38. Explain R's scoping rules and copy-on-modify at the level you need for clinical work.

SHORT ANSWER. R uses lexical scoping: a function looks for a name in its own body first, then in the environment where the function was defined, and onwards up to the global environment and the loaded packages. Copy-on-modify means that when you assign an object to a second name and then change one of them, R makes a copy — so a function can never accidentally modify its caller's data.

FOR THE SAS PROGRAMMER. The DATA step modifies the PDV in place and the effect is visible immediately, and a macro variable's scope depends on where it was defined with global and local rules that surprise people. R's copy-on-modify is the safety property you never had: passing ADSL to a function cannot change the caller's ADSL. The exception is data.table, which is exactly why data.table needs a comment when you use it.

DETAIL. What you need to state: R does not copy on assignment, it copies on the first modification, and since R 3.5 it tracks references so it often avoids the copy entirely when it can prove the object has one name. The practical consequences are that memory can spike when you modify a large object, and that a function has no way to change its argument unless you explicitly return the changed value. The `<<-` operator assigns into an enclosing environment and is the exception; it is legal, occasionally useful for a counter, and a red flag in review because it makes data flow invisible. Environments themselves have reference semantics, which is what R6 classes and data.table build on.

```
a <- c(1, 2, 3)
b <- a                              # no copy yet -- both names, one object
b[1] <- 99                          # NOW it copies
a                                   # 1 2 3  -- untouched

# functions cannot modify their arguments
add_flag <- function(df) { df$FLAG <- "Y"; df }
adsl <- data.frame(USUBJID = "01-001")
add_flag(adsl)                      # returns a df with FLAG
names(adsl)                         # "USUBJID" -- the caller is unchanged

# lexical scoping: defined-where, not called-from
x <- "global"
outer <- function() { x <- "outer"; inner() }
inner <- function() x
outer()                             # "global" -- inner was DEFINED at top level

# tracemem shows the copy happening
v <- c(1, 2, 3); tracemem(v); v[1] <- 9; untracemem(v)
```

INTERVIEW TRAP. "So how does data.table fit into this?" data.table's `:=` deliberately breaks copy-on-modify: it modifies the object by reference, so a function can change the caller's table. That is the source of its speed and its most surprising behaviour, and calling `copy()` is how you opt back into the safe semantics. Being able to connect those two answers — the general rule and the one deliberate exception — is what a senior answer sounds like.

Section 16 — Debugging and Error Messages

Q39. How do you debug R code?

SHORT ANSWER. `traceback()` after an error shows the call stack that led to it. `browser()` inserted in a function pauses execution there and drops you into an interactive prompt with the local variables live. `debugonce(f)` does the same without editing the function. `print()` still works and is often fastest.

FOR THE SAS PROGRAMMER. SAS debugging is reading the log — NOTEs, the number of observations, `PUT _ALL_` to dump the PDV. R's equivalent of `PUT _ALL_` is `browser()`, and it is far better: you are not printing a snapshot, you are standing inside the function with everything in scope and a live prompt. That is the single biggest quality-of-life upgrade R gives you over the SAS log.

DETAIL. Inside `browser()` the commands are `n` for the next statement, `s` to step into a call, `c` to continue, `f` to finish the current loop or function, and `Q` to quit. `options(error = recover)` drops you into a frame chooser on any error, which is the fastest way to diagnose something deep in a pipeline. For tidyverse errors, `rlang::last_trace()` gives a much more readable stack than `traceback()`. In RStudio, you can set a breakpoint by clicking the left margin (with the file sourced), and "Rerun with Debug" replays a failing script. `sessionInfo()` belongs in every log for reproducibility, and in a regulated environment `renv` pins the package versions so the debug you did yesterday still reproduces.

```
derive_bmi <- function(data) {
  # browser()                          # uncomment to pause here
  data |> dplyr::mutate(BMI = WEIGHT / (HEIGHT^2))
}

adsl <- data.frame(USUBJID = "01-001", WEIGHT = 70, HEIGHT = "1.75")  # note: character!
try(derive_bmi(adsl))                 # non-numeric argument to binary operator
traceback()                           # where it happened

debugonce(derive_bmi)                 # step through the next call only
# derive_bmi(adsl)

options(error = recover)              # drop into a frame chooser on any error
options(error = NULL)                 # switch it back off

# minimum reproducibility hygiene
sessionInfo()
```

INTERVIEW TRAP. "How do you debug inside a `map()` or a long pipe?" `browser()` inside the anonymous function works, and `purrr::map()` has a `.progress` argument for long runs, but the practical answer is to pull the body out into a named function, test it on one element, and only then map it. Interviewers ask this because debugging inside functional code is where beginners get stuck.

Q40. What do these errors actually mean: "object of type 'closure' is not subsettable", "argument is of length zero", and the recycling warning?

SHORT ANSWER. "Closure is not subsettable" means you tried to index a function — almost always because the object name you used does not exist, so R found a base function of the same name instead. "Argument is of length zero" means an `if` received something empty, usually because a

lookup returned nothing. The recycling warning means two vectors of incompatible lengths met in an operation.

FOR THE SAS PROGRAMMER. These are R's equivalents of the SAS log messages you scan for, and like SAS's, the message names the symptom rather than the cause. "Closure is not subsettable" is roughly SAS's "variable not found" wearing a disguise, because R fell back to a function of that name instead of telling you the variable is missing.

DETAIL. For the closure error: `data`, `df`, `c`, `mean`, `T` and `length` are all bound to base R objects, so `data$AGE` when you never created `data` finds `utils::data`, the function, and complains about subsetting a closure. The fix is to check the object exists with `exists()` and to avoid naming datasets `data` or `df`. For "argument is of length zero": `if (x == 1)` where `x` is `character(0)` or `NULL`. Related and distinct is "missing value where TRUE/FALSE needed", which is `if (NA)` — a missing value in a condition, the classic clinical bug. Since R 4.2.0, a condition of length greater than one is a hard error ("the condition has length > 1") rather than the old silent use-the-first-element behaviour.

```
# 1. object of type 'closure' is not subsettable
rm(list = ls())
try(data$AGE)          # Error: object of type 'closure' is not subsettable
                       # cause: `data` was never created; R found utils::data
exists("data", inherits = FALSE)      # FALSE -- the real diagnosis

# 2. argument is of length zero
x <- character(0)
try(if (x == "Y") 1)   # Error in if (x == "Y") 1 : argument is of length zero
if (length(x) == 1 && x == "Y") 1     # guard on length first

# 3. missing value where TRUE/FALSE needed  -- the clinical one
age <- NA
try(if (age < 65) "young")            # Error: missing value where TRUE/FALSE needed
if (!is.na(age) && age < 65) "young"  # && short-circuits, so this is safe

# 4. condition has length > 1 (an ERROR since R 4.2.0)
try(if (c(TRUE, FALSE)) 1)            # Error: the condition has length > 1

# 5. the recycling warning
c(1, 2, 3) + c(10, 20)
# Warning: longer object length is not a multiple of shorter object length

# 6. $ operator is invalid for atomic vectors
v <- c(AGE = 45)
try(v$AGE)                            # use v[["AGE"]] -- $ works on lists, not vectors
```

INTERVIEW TRAP. The follow-up on the missing-value error is "why does `&&` fix it?" Because `&&` short-circuits: it evaluates the left side first and, if that is FALSE, never evaluates the right. So `!is.na(age) && age < 65` never compares NA. Note that `&&` requires length-one operands (an error since R 4.3.0 if you give it a vector), whereas `&` is the vectorised version for use inside `filter()` and `mutate()`. Using `&&` inside a `filter()` is a common and quietly wrong habit.

Module 2 — pharmaverse End-to-End: SDTM, ADaM, TLF, and R-Based Submission

This module covers the open-source R stack that clinical programming teams actually use to go from raw/SDTM through ADaM, TLFs, and an eCTD package — admiral, metacore/metatools, xportr, rtables/Tplyr, and the R Consortium FDA submission pilots. Every function name and argument here was verified against installed packages on 2026-08-02 (admiral 1.4.1.9061, rtables 0.6.15.9005, xportr 0.5.0, metacore 0.3.0, metatools 0.3.0, formatters 0.5.12, tern 0.9.10, r2rtf 1.3.0) or fetched from the package source; where a package is not installed here the code block is labelled illustrative. A reality check before you memorise all of it: of the senior R clinical roles surveyed for this pack, only one named pharmaverse explicitly (admiral and tidyCDISC, listed as preferred), so treat this as a strong differentiator rather than a checklist — know it deeply, but lead with the concepts, not the package names.

Section 1 — What pharmaverse actually is

Q1. What is pharmaverse, in your own words?

SHORT ANSWER. pharmaverse is not a package and not a product — it's a curated, cross-company network of open-source R packages for clinical reporting, from CRF to eSubmission, plus a lightweight governance layer that decides what gets listed. The packages are owned and maintained by different companies (Roche, GSK, Novo Nordisk, Atorus, Merck, Novartis and others), not by a central vendor. The site itself describes it as "a connected network of companies and individuals working to promote collaborative development of curated open source R packages."

FOR THE SAS PROGRAMMER. The closest analogue is not SAS itself — it's the shared macro library your company built over 15 years, except open-sourced and split across a dozen sponsors. There is no `SAS/STAT` here: no single vendor, no single support contract, no guaranteed backward compatibility across the whole set. What you gain is that the "macro library" was written once by the industry instead of separately by every sponsor, so a new hire already knows it.

DETAIL. pharmaverse organises packages by end-to-end stage — SDTM, ADaM, TLG, eSub, plus metadata and utility categories. The current listing runs to roughly 48 packages, including admiral, sdtm.oak, metacore, metatools, xportr, rtables, tern, rlistings, Tplyr, tfrmt, tidytlg, r2rtf, datasetjson, pkglite, logrx, whirl, valtools, riskmetric, risk.assessr, teal, tidyCDISC, siera, connector, envsetup, and the test-data packages pharmaversesdtm / pharmaverseadam / pharmaverseraw. Packages are distributed on CRAN and also on an R-universe CRAN-like server, with the explicit expectation that production users maintain their own internal repository snapshot rather than installing from the internet.

```
# The end-to-end stack, as it actually loads in a study repo
library(dplyr)
library(admiral)        # ADaM derivations
library(metacore)       # the spec as a first-class object
library(metatools)      # apply the spec to data
library(xportr)         # write submission-ready XPT v5
library(rtables)        # TLF layouts

packageVersion("admiral")
```

INTERVIEW TRAP. "Is pharmaverse validated?" No. The pharmaverse site carries an explicit disclaimer that listing implies no endorsement of code reliability and that the packages are not validated, not GxP-assured, and not regulatory-assured — users must assess suitability themselves. Anyone who says "it's validated because it's pharmaverse" has just failed the question. The correct answer is that qualification is the sponsor's job and points at the R Validation Hub approach: risk assessment (riskmetric, risk.assessr), an internal validated repository, and documented installation qualification.

Q2. How does pharmaverse governance work — how does a package get in?

SHORT ANSWER. It's deliberately lightweight. Anyone raises an issue on the pharmaverse GitHub repo using the "New package request" template, the proposal goes to the community Slack for comment, and if nobody objects the website team actions it. Contentious decisions escalate to a council. Removal follows the same route — anyone can request removal with rationale.

FOR THE SAS PROGRAMMER. There is no equivalent of SAS Institute's release QA gate. Inclusion means "the community didn't object," not "this passed validation." Compare it to how a standards team accepts a contributed macro into the library: a review, a conversation, a decision — but the burden of proving fitness for a given study still sits with the study team.

DETAIL. There is a published Charter covering objectives and scope and recommendations for new collaborations; a council as an escalation body; a website team for routine actions; and the broader Slack community for proposals. Packages carry category tags (adam, sdtm, tlg, utilities, validation) rather than quality badges. Notably there is no formal badging or acceptance-criteria checklist — the signals of quality are indirect: is it on CRAN, is there a test suite and coverage badge, is there an active maintainer, does the maintaining company use it in production. That is precisely the assessment riskmetric and risk.assessr try to systematise.

```
# What you actually check before adopting a package, in order
# 1. Is it on CRAN (implies R CMD check passes on multiple platforms)?
available.packages()["admiral", c("Version", "Repository")]

# 2. Reverse-check what it drags in — dependency surface is risk surface
tools::package_dependencies("admiral", recursive = FALSE)[[1]]
```

INTERVIEW TRAP. The follow-up is "so how would you decide to use admiral for a submission study?" Don't answer "because everyone uses it." Answer with a risk-based framework: dependency footprint, test coverage, maintainer responsiveness, whether the specific functions you use are covered by unit tests, whether you can pin the version in renv, and whether you run your own qualification suite against a known-answer dataset. Then note that admiral scores well on all of those — CRAN-released, heavily tested, corporate-backed — which is why it's the default, not the other way round.

Q3. What's the risk of treating pharmaverse as one product?

SHORT ANSWER. Version skew and paradigm collision. The packages release independently on different cadences, they don't all share a data model, and some of them are mutually exclusive design philosophies rather than complementary layers. If you assume "pharmaverse just works together," you'll discover mid-study that Tplyr's output shape and rtables' table object don't compose, or that an admiral extension pinned to admiral 1.3 breaks under 1.5.

FOR THE SAS PROGRAMMER. Your macro library had one owner and one release. Here you have a dozen owners. It's closer to a study where every function area supplied its own macro suite and nobody agreed on the parameter naming convention.

DETAIL. Three concrete failure modes. First, **release skew**: admiral releases in two phases — Phase 1 is admiraldev, pharmaversesdtm, admiral core; Phase 2 is the therapeutic-area extensions plus pharmaverseadam — and the extensions release on a content-driven, sometimes ad hoc schedule, so an extension can lag the core by months. Second, **paradigm collision**: rtables builds a structured table object with addressable row/column paths; Tplyr builds a tidy data frame you then render yourself. Both are excellent; picking both for one study means two rendering pipelines and two QC approaches. Third, **overlapping scope**: gtsummary, tern, Tplyr, tidytlg and tfrmt all summarise clinical data, with different opinions. Choosing is an architecture decision, not a preference.

```
# The only real defence: pin everything, once, at study setup
# renv snapshots the exact versions into renv.lock, checked into the repo
library(renv)
# renv::init()      # at study start
# renv::snapshot()  # after the stack is agreed
# renv::restore()   # every other programmer, every rerun, every reviewer
```

INTERVIEW TRAP. "How do you handle a breaking change in admiral mid-study?" The answer is that you don't — you froze the version at study setup via renv.lock plus an internal repository snapshot, and an upgrade is a change-controlled event with a re-run and a comparison, not a `install.packages()`. admiral does help: it uses lifecycle deprecation with a roughly three-year cycle, so functions warn before they're removed. But the control has to be yours.

Q4. Walk me through the packages end to end for one study.

SHORT ANSWER. Spec in metacore, raw-to-SDTM in sdtm.oak (or bespoke tidyverse), SDTM-to-ADaM in admiral with metatools applying the spec, TLFs in rtables/tern or Tplyr, RTF via r2rtf or the formatters exporters, XPT v5 via xportr, and the eCTD package assembled with pkglite plus a define.xml and an ADRG.

FOR THE SAS PROGRAMMER. Map it onto what you do now: the spec workbook stays a spec workbook, but it becomes machine-readable and drives the code instead of sitting beside it. The `%make_adam` style macro becomes a pipeline of small named derivations. `PROC REPORT` becomes rtables or Tplyr. `PROC EXPORT` / `LIBNAME XPORT` becomes xportr. `PROC COMPARE` becomes diffdf.

DETAIL. The pharmaverse-canonical chain is: **Metadata → OAK → admiral → define.xml → TLGs → Submission.** In practice most sponsors enter at SDTM (the CRO or the EDC vendor already produced it) and the R work starts at ADaM. The metadata layer is the piece SAS programmers most often underestimate: in the R workflow the spec is loaded once as a metacore object and then used — to select variables, to build derived variables from codelists, to check controlled terminology, to order and label columns, and finally to type and length the XPT. That is the biggest single behavioural difference from a SAS shop where the spec is a Word/Excel document a human reads.

```
library(metacore)
library(metatools)
library(admiral)
library(xportr)
library(dplyr)

# 1. Spec becomes an object
spec_path <- metacore_example("p21_mock.xlsx")
mc <- spec_to_metacore(spec_path, quiet = TRUE)
adsl_spec <- select_dataset(mc, "ADSL")

# 2. Pull the predecessor variables the spec says come from SDTM
#    (build_from_derived reads the spec's derivation column)
# adsl_pred <- build_from_derived(adsl_spec, list(dm = dm), predecessor_only = FALSE)

# 3. admiral derivations ... 4. metatools checks ... 5. xportr writes
```

INTERVIEW TRAP. They may ask where the analysis results live. The newer answer is ARDs — Analysis Results Datasets, standardised by CDISC ARS — with the `cards` / `cardx` packages producing them and `siera` generating ARD programs from ARS metadata. If you mention ARS/ARD as "the direction of

travel, the standard was released in 2024 and tooling is early," you sound current. Don't claim it's widely deployed; it isn't.

Q5. Honestly — would you propose pharmaverse for our next submission study?

SHORT ANSWER. For ADaM and TLFs, yes, with a pinned environment and a qualification step. For SDTM, I'd be cautious — the R tooling there is genuinely less mature. And I'd be explicit that "R-based submission" means the datasets are still XPT v5 and the define.xml still probably comes from Pinnacle 21, because that's the honest state of the tooling.

FOR THE SAS PROGRAMMER. This is the answer that separates an enthusiast from someone you'd trust with a filing. The differentiator isn't knowing the package names; it's knowing where the stack stops.

DETAIL. The defensible position: admiral is production-grade — CRAN-released, heavily tested, used in real filings, with an explicit design philosophy of no black boxes. rtables and tern are production-grade and were used in the R Consortium pilots. xportr does the XPT job well. The soft spots are (a) SDTM automation, where sdtm.oak is young and the work is inherently study-specific; (b) define.xml generation, where the only open-source R option, defineR, is explicitly experimental, supports only define 2.0.0, and its own authors say it is not recommended for regulatory submission yet; and (c) the ADRG/SDRG, which are still Word documents a human writes. A credible plan is: R for ADaM and TLF, existing tooling for SDTM and define, dual programming for the first study, and a documented environment.

```
# The three artefacts that make an R study auditable, all cheap:
# 1. renv.lock            - exact package versions
# 2. sessionInfo() dump   - R version, OS, locale (locale matters for sorting!)
# 3. a run log per program
si <- sessionInfo()
si$R.version$version.string
Sys.getlocale("LC_COLLATE")   # character sort order differs by locale
```

INTERVIEW TRAP. If you oversell, the next question is "which parts have you actually done end to end?" Have a truthful boundary ready. Saying "I've built ADSL and BDS in admiral and produced RTF outputs; I have not personally generated a define.xml from R" is far stronger than a vague claim, because the interviewer probably hasn't either.

Section 2 — admiral: design philosophy and the derivation grammar

Q6. What is admiral's design philosophy, and why does it matter?

SHORT ANSWER. Four stated principles: usability, simplicity, findability, readability. Concretely: modular functions with one clear purpose, no black boxes, tidyverse-style pipelines, and code that a health-authority reviewer who doesn't know R can still follow. admiral explicitly does not try to derive a whole ADaM dataset for you — it gives you the building blocks.

FOR THE SAS PROGRAMMER. The opposite of `%make_adsl(study=XYZ)`. A macro that produces a whole dataset hides its logic in a 900-line file nobody reads; admiral forces the logic to be visible in the study program as a sequence of named steps. Your spec-to-code mapping becomes one-to-one, which is exactly what a reviewer wants and exactly what makes the code longer.

DETAIL. From the package's own principles: each function has a clear purpose; arguments shape how an output is computed but never change what the function is for; similar tasks are consolidated (one study-day function, not one function per `--DY` variable); nesting three or four functions deep should be exceptional; and the docs state modules should be "understandable by outside readers including reviewers at health authorities." The practical consequence is that an admiral ADSL program is longer than a SAS macro call and shorter than the macro. You trade call-site brevity for total transparency. Also important: admiral will not silently guess. Where the ADaM IG leaves a choice to the SAP — imputation rules, which record is the baseline, what counts as treatment-emergent — admiral makes you state it as an argument.

```
library(admiral)
library(dplyr)

# admiral ships runnable templates - the fastest way to learn the idiom
list_all_templates()          # e.g. ad_adsl, ad_advs, ad_adae, ad_adtte
# use_ad_template("ADSL")     # writes a working starter program to your project
```

INTERVIEW TRAP. "Isn't all that boilerplate a step backwards from a macro library?" The strong answer names the trade explicitly: a macro library optimises for the programmer writing the 40th study; admiral optimises for the reviewer, the validator, and the person who inherits the study. And the boilerplate is where a company layer goes — most sponsors wrap admiral in a thin internal package of standard derivations, which is exactly what the "company extensions" like admiralroche and admiralgsk are.

Q7. Explain the admiral naming families — derive_var_, derive_vars_, derive_param_.

SHORT ANSWER. `derive_var_*` adds one variable; `derive_vars_*` adds several; `derive_param_*` adds rows — new records for a derived parameter in a BDS dataset. There are also `compute_*` functions that work on vectors rather than datasets, and `filter_*` functions for record selection.

FOR THE SAS PROGRAMMER. Singular/plural is the "how many columns" signal. `derive_param_*` is the DATA step that outputs extra observations with a new PARAMCD — the SAS pattern of `set adlb; ...; paramcd = 'BMI'; output;`. `compute_*` are the pure functions you'd have written as a `%sysfunc`-able

expression, and they're deliberately exported so you can use them inside your own `mutate()` when admiral's dataset-level wrapper doesn't fit.

DETAIL. The families and their reference keywords: `der_gen` (all ADaMs), `der_adsl`, `der_bds_findings`, `der_occds` for variable derivations; `der_prm_bds_findings` and `der_prm_tte` for parameter derivations; `com_date_time` and `com_bds_findings` for computations. Knowing the split matters because it tells you which function you want without searching: if the answer is "a new row," you want a `derive_param_*` or `derive_summary_records()` / `derive_extreme_records()`; if it's "a new column," a `derive_var(s)_*`.

```
library(admiral)
library(dplyr)

# compute_* works on vectors - usable anywhere, including inside mutate()
compute_duration(
  start_date = as.Date("2023-01-01"),
  end_date   = as.Date("2023-01-31")
)                                  # 31 days by default (add_one = TRUE)

compute_bmi(height = 180, weight = 75)
compute_egfr(creat = 90, creatu = "umol/L", age = 55, weight = 75, sex = "M", method = "CRCL")
```

INTERVIEW TRAP. A classic mis-answer is claiming `derive_var_ablfl()` exists. It doesn't. Nor does `derive_var_anl01fl()`. admiral deliberately has no per-ADaM-variable function for flags whose definition is SAP-dependent — you use `derive_var_extreme_flag()` and tell it the rule. If you name a function you're not certain of, say "the extreme-flag function" and describe the arguments; a wrong function name is the single fastest way to lose a technical interviewer.

Q8. Explain the dataset_add join grammar — by_vars, order, mode, filter_add.

SHORT ANSWER. `derive_vars_merged()` is admiral's workhorse many-to-one merge: `dataset_add` is the dataset you're pulling from, `filter_add` subsets it before the join, `order` plus `mode = "first"/"last"` collapses it to one record per `by_vars`, and `new_vars` names exactly which columns come across and what they're renamed to. Nothing is joined implicitly.

FOR THE SAS PROGRAMMER. This is `PROC SORT` + a DATA step `MERGE ... BY ... ; IF first.usubjid;` compressed into one call — but with a crucial difference: SAS's MERGE will silently produce a many-to-many cartesian mess and just carry the last value. `derive_vars_merged()` **errors** if the join isn't many-to-one after `filter_add` and `order`/`mode` are applied. That error has caught real bugs that survived years of SAS production.

DETAIL. Argument by argument. `dataset_add` — the source. `by_vars` — the join keys, always as `exprs()`. `filter_add` — a condition applied to `dataset_add` only; this is where "only dosing records with a positive dose" lives. `order` + `mode` — how to pick one record when several remain. `new_vars` — `exprs(TRTSDTM = EXSTDTM)` both selects and renames; omit it and you get everything, which you almost never want. `missing_values` — what to set for subjects with no matching record, which is how you avoid silent `NA`. When a merge fails on multiplicity, `get_many_to_one_dataset()` returns the offending records so you can look at them immediately.

```
library(admiral)
library(dplyr)
library(stringr)

adsl <- adsl %>%
  derive_vars_merged(
    dataset_add = ex_ext,
    filter_add  = (EXDOSE > 0 | (EXDOSE == 0 & str_detect(EXTRT, "PLACEBO"))) &
                  !is.na(EXSTDTM),
    new_vars    = exprs(TRTSDTM = EXSTDTM, TRTSTMF = EXSTTMF),
    order       = exprs(EXSTDTM, EXSEQ),
    mode        = "first",
    by_vars     = exprs(STUDYID, USUBJID)
  )

# When it errors on multiplicity, inspect what caused it:
# get_many_to_one_dataset()
```

INTERVIEW TRAP. "Where does the filter go?" Filtering `dataset_add` with `filter_add` and filtering the result are different operations, and the wrong one silently drops subjects from ADSL. Rule of thumb: if the condition is about the record you're pulling from (dose > 0), it's `filter_add`; if it's about the subject staying in the output at all, it belongs upstream in the ADSL population, not in the merge. Second trap: forgetting `missing_values` and then wondering why `TRTSDT` is `NA` for screen failures — which is correct, but only if you decided it deliberately.

Q9. When would you use derive_vars_joined instead of derive_vars_merged?

SHORT ANSWER. When the record you want depends on a condition that compares the two datasets — "the last dose before this AE started," "the lab value on or after the visit window opens." `derive_vars_merged` filters only the source dataset; `derive_vars_joined` lets you filter the joined pair with `filter_join`, using `join_vars` to bring the comparison columns into scope.

FOR THE SAS PROGRAMMER. `derive_vars_merged` is a DATA step MERGE. `derive_vars_joined` is a `PROC SQL` self-join with a correlated condition, or the DOW-loop you wrote to find last dose prior to onset. In SAS this is the derivation that always takes an hour and always has an off-by-one on the equality; here it's one call with the condition written out literally.

DETAIL. The key arguments beyond the merged set: `join_vars` — variables from `dataset_add` you need visible in the join condition but may not want in the output; `join_type` — `"all"`, `"before"`, `"after"` relative to `order`; `filter_join` — the condition evaluated on the joined pair, where both sides' columns are in scope; `first_cond_lower` / `first_cond_upper` — restrict the joined window to records after/before the first record meeting a condition. Then `order` + `mode` still collapse to one. There is also `derive_vars_joined_summary()` when you want an aggregate over the joined window (e.g. cumulative dose before the event) rather than a single record.

```
library(admiral)
library(dplyr)

# Last dose taken at or before AE onset
adae <- derive_vars_joined(
  adae,
  dataset_add = ex_single,
  by_vars     = exprs(STUDYID, USUBJID),
  new_vars    = exprs(LDOSEDTM = EXSTDTM, LDOSE = EXDOSE),
  join_vars   = exprs(EXSTDTM),
  join_type   = "all",
  order       = exprs(EXSTDTM),
  filter_join = EXSTDTM <= ASTDTM,
  mode        = "last"
)
```

INTERVIEW TRAP. The interviewer may ask what happens to subjects with no qualifying dose record. They get `NA` unless you set `missing_values`, and — this is the subtle one — `derive_vars_joined` will not drop them; the subject stays with missing new variables. That's usually right, but it means a downstream `filter(LDOSE > 0)` silently removes them. Second trap: `join_type = "before"` / `"after"` are relative to `order`, so getting `order` wrong flips the meaning of the window without any error.

Q10. What are restrict_derivation, slice_derivation and call_derivation for?

SHORT ANSWER. They're higher-order functions: they take a derivation function plus its arguments and change when or how many times it runs. `restrict_derivation` runs a derivation only on records matching a filter but keeps the rest; `slice_derivation` runs it with different arguments on different slices of the data; `call_derivation` runs the same derivation repeatedly with varying arguments.

FOR THE SAS PROGRAMMER. `restrict_derivation` is "do this logic only `if trtemfl='Y'` but don't drop the other rows" — in SAS you'd either subset and re-merge, or wrap the whole block in an `IF`. `slice_derivation` is the case you meet constantly with date imputation: impute AE start differently for pre-treatment and post-treatment records. `call_derivation` is a `%do` loop over a macro call — the SAS macro-loop that generates `AVAL`, `AVALC`, ... for a list of variables.

DETAIL. `restrict_derivation(derivation = , args = params(...), filter = )` is the important one, and it is how ABLFL and the AOCCxxFL occurrence flags are actually derived — you flag the extreme record only within the eligible subset. `slice_derivation(derivation = , args = params(...), derivation_slice(filter = , args = params(...)), ...)` applies the first matching slice, so slice order matters and the last slice is your catch-all. `call_derivation(derivation = , variable_params = list(params(...), params(...)), <shared args>)` factors the shared arguments out.

```
library(admiral)
library(dplyr)

# ABLFL: last non-missing value on or before treatment start, per parameter
advs <- restrict_derivation(
  advs,
  derivation = derive_var_extreme_flag,
  args = params(
    by_vars = exprs(STUDYID, USUBJID, BASETYPE, PARAMCD),
    order   = exprs(ADT, ATPTN, VISITNUM),
    new_var = ABLFL,
    mode    = "last"
  ),
  filter = (!is.na(AVAL) & ADT <= TRTSDT & !is.na(BASETYPE))
)

# Different imputation rule pre- vs post-treatment
ae <- slice_derivation(
  ae,
  derivation = derive_vars_dt,
  args = params(dtc = AESTDTC, new_vars_prefix = "AST"),
  derivation_slice(filter = AESTDTC >= TRTSDTC,
                   args = params(date_imputation = "first", highest_imputation = "M")),
  derivation_slice(filter = TRUE,
                   args = params(highest_imputation = "n"))
)
```

INTERVIEW TRAP. People confuse `restrict_derivation`'s `filter` with `filter_add`. `filter_add` subsets the source dataset inside a merge; `restrict_derivation`'s `filter` decides which output records the derivation is applied to, leaving the others untouched rather than removed. Say that distinction out loud and you've demonstrated you've actually used both.

Q11. Contrast the ADSL, BDS and OCCDS patterns in admiral.

SHORT ANSWER. ADSL is one row per subject built by repeated merges from SDTM — every derivation adds columns. BDS is one row per subject per parameter per timepoint, built by stacking parameters and then deriving analysis variables — some derivations add rows. OCCDS is one row per occurrence (AE, CM, MH) with occurrence flags and dictionary/query variables, and the hard part is flagging, not computing.

FOR THE SAS PROGRAMMER. Structurally identical to what you already build; what changes is that admiral gives each pattern its own function family and its own vignette, and the templates (`use_ad_template()`) encode the canonical order of operations. The order matters more in R than in SAS because each step is a pure function of the previous data frame — you can't reach back to a variable you haven't created yet.

DETAIL. **ADSL**: dates first (`derive_vars_dt` / `derive_vars_dtm` on `--DTC`), then treatment variables from EX by merge, then `derive_vars_dtm_to_dt` and `derive_var_trtdurd`, then demographics/age (`derive_vars_aage`), grouping (`derive_vars_cat`), death and last-known-alive via `derive_vars_extreme_event`, then population flags via `derive_var_merged_exist_flag`. **BDS**: start from the SDTM findings domain, map `--TESTCD` to `PARAMCD` with `derive_vars_merged_lookup`, derive `ADT`/`ADY`, set `AVISIT`/`AVISITN`, derive `AVAL`, add derived parameters with `derive_param_computed` / `derive_summary_records`, then `BASETYPE` (`derive_basetype_records`), `ABLFL`, `BASE`, `CHG`, `PCHG`, `ANRIND`, and finally the analysis flags. **OCCDS**: merge ADSL variables, impute and derive `ASTDT`/`AENDT`,

`TRTEMFL`, dictionary-derived queries via `derive_vars_query`, then occurrence flags with `restrict_derivation` + `derive_var_extreme_flag`, and `derive_var_obs_number` for `ASEQ`.

```
library(admiral)
library(dplyr)

# OCCDS: bring ADSL vars in without listing them twice
adsl_vars <- exprs(TRTSDT, TRTEDT, TRT01A, TRT01P, DTHDT)

adae <- ae %>%
  derive_vars_merged(
    dataset_add = adsl,
    new_vars    = adsl_vars,
    by_vars     = exprs(STUDYID, USUBJID)
  )

# ... and later drop them again before output, or re-merge the rest:
# derive_vars_merged(dataset_add = select(adsl, !!!negate_vars(adsl_vars)), by_vars = ...)
```

INTERVIEW TRAP. The BDS ordering trap: `BASE` must come after `ABLFL`, `CHG` after `BASE`, and `derive_basetype_records()` before `ABLFL` if you have multiple baseline definitions (e.g. pre-dose per period in a crossover). Get the order wrong and you get `NA`s, not errors. Interviewers love asking "what order do you derive ABLFL, BASE and CHG in, and why?"

Q12. What are the admiral extension packages, and when do you reach for one?

SHORT ANSWER. admiral core is therapeutic-area agnostic. TA extensions add derivations that only make sense in one area: admiralonco (RECIST response, progression-free survival endpoints), admiralophtha, admiralvaccine, admiralpeds, admiralmetabolic, admiralneuro. There are also company extensions — admiralroche, admiralgsk are the named examples — which hold company-specific plumbing like metadata access.

FOR THE SAS PROGRAMMER. Same structure as a corporate macro library: a global standards layer, a TA layer, and a company layer. The difference is the first two are open source and shared across sponsors.

DETAIL. The three-layer model is explicit in admiral's documentation: a core package usable by any company, TA extensions, and company extensions. Features that turn out to be relevant across several extensions migrate into the core — which is why you should check the core reference before assuming something lives in an extension. Release cadence: admiral core, admiraldev and pharmaversesdtm release together in "Phase 1"; extensions plus pharmaverseadam follow in "Phase 2," on a content-driven and sometimes ad hoc schedule. That gap is the source of most version-pinning pain.

```
#| skip-check
# Illustrative only - admiralonco is not installed in this environment.
# The oncology extension supplies response/progression source objects that
# feed the core derive_param_tte() machinery:
library(admiralonco)
# Check the current admiralonco reference for exact function and argument
# names before quoting them - the oncology API has moved between releases.
```

INTERVIEW TRAP. Do not bluff extension function names. The oncology API in particular has changed across releases. The credible answer is structural: "admiralonco supplies the oncology-specific event and censoring source objects and response derivations, and they feed the same `derive_param_tte()` in the core — I'd check the current reference for exact names." That shows you understand the architecture, which is what's being tested.

Section 3 — Worked ADaM derivations, and the SAS comparison

Q13. Derive treatment start and end dates in admiral. How is it different from SAS?

SHORT ANSWER. Impute the EX datetimes first, then two `derive_vars_merged()` calls against the exposure dataset — first dosing record for `TRTSDTM`, last for `TRTEDTM` — with `filter_add` restricting to actual dosing records. Then `derive_vars_dtm_to_dt()` for the date versions and `derive_var_trtdurd()` for duration.

FOR THE SAS PROGRAMMER. In SAS this is a sort of EX by USUBJID EXSTDTC, a `where` on dose, and `first./last.` processing, then a merge back to ADSL. Same shape. The differences are that the placebo-zero-dose condition is written inline where a reviewer can see it instead of buried in a macro, and that the imputation of a partial `EXSTDTC` is an explicit argument rather than an assumption.

DETAIL. The canonical order is: `derive_vars_dtm()` on `EXSTDTC` and `EXENDTC` (end date imputed to last time), then the two merges with `order = exprs(EXSTDTM, EXSEQ)` / `mode = "first"` and `mode = "last"`, then `derive_vars_dtm_to_dt(source_vars = exprs(TRTSDTM, TRTEDTM))`, then `derive_var_trtdurd()` which takes no arguments and computes `TRTDURD` from `TRTSDT` / `TRTEDT` inclusive of both endpoints. Note `derive_vars_dtm()` also emits `EXSTTMF`, the time imputation flag, which you carry through as `TRTSTMF` — because ADaM requires you to flag imputation.

```
library(admiral)
library(dplyr)
library(stringr)

ex_ext <- ex %>%
  derive_vars_dtm(dtc = EXSTDTC, new_vars_prefix = "EXST") %>%
  derive_vars_dtm(dtc = EXENDTC, new_vars_prefix = "EXEN", time_imputation = "last")

adsl <- adsl %>%
  derive_vars_merged(
    dataset_add = ex_ext,
    filter_add  = (EXDOSE > 0 | (EXDOSE == 0 & str_detect(EXTRT, "PLACEBO"))) & !is.na(EXSTDTM),
    new_vars    = exprs(TRTSDTM = EXSTDTM, TRTSTMF = EXSTTMF),
    order       = exprs(EXSTDTM, EXSEQ),
    mode        = "first",
    by_vars     = exprs(STUDYID, USUBJID)
  ) %>%
  derive_vars_merged(
    dataset_add = ex_ext,
    filter_add  = (EXDOSE > 0 | (EXDOSE == 0 & str_detect(EXTRT, "PLACEBO"))) & !is.na(EXENDTM),
    new_vars    = exprs(TRTEDTM = EXENDTM, TRTETMF = EXENTMF),
    order       = exprs(EXENDTM, EXSEQ),
    mode        = "last",
    by_vars     = exprs(STUDYID, USUBJID)
  ) %>%
  derive_vars_dtm_to_dt(source_vars = exprs(TRTSDTM, TRTEDTM)) %>%
  derive_var_trtdurd()
```

INTERVIEW TRAP. "Why `EXDOSE == 0 & str_detect(EXTRT, 'PLACEBO')`?" Because a placebo record legitimately has zero dose and still counts as treatment — filtering on `EXDOSE > 0` alone silently gives placebo subjects a missing `TRTSDT`, which then propagates into `SAFFL`, `TRTEMFL` and `ADY`. That's a real, findable bug and a favourite interview question.

Q14. Derive study day. What's the gotcha?

SHORT ANSWER. `derive_vars_dy(reference_date = TRTSDT, source_vars = exprs(ADT))`. The gotcha is that there is no day zero: dates on or after the reference give `date - ref + 1`, dates before give `date - ref`, so day −1 is followed directly by day 1. admiral implements this correctly; hand-rolled arithmetic usually doesn't.

FOR THE SAS PROGRAMMER. Exactly the same CDISC rule you already apply, and exactly the same place people get it wrong. In SAS it's `if adt >= trtsdt then ady = adt - trtsdt + 1; else ady = adt - trtsdt;`. The advantage in admiral is that it derives `--DY` for every date variable you list at once, and names the outputs by convention.

DETAIL. `derive_vars_dy()` takes a `reference_date` and a set of `source_vars`; it derives one `--DY` per source variable using the variable's own prefix, so `exprs(ADT, ASTDT, AENDT)` yields `ADY`, `ASTDY`, `AENDY`. You can also override the output name inside `exprs()`. Because R `Date` arithmetic returns a `difftime`, admiral coerces to integer for you — worth knowing, because if you compute study day yourself with `as.numeric(ADT - TRTSDT)` you get a numeric that will write to XPT as a double with a decimal format unless you fix it.

```
library(admiral)
library(dplyr)

adae <- adae %>%
  derive_vars_dy(
    reference_date = TRTSDT,
    source_vars    = exprs(ASTDT, AENDT)
  )   # -> ASTDY, AENDY

advs <- advs %>%
  derive_vars_dy(reference_date = TRTSDT, source_vars = exprs(ADT))  # -> ADY
```

INTERVIEW TRAP. Two follow-ups. First: which reference date — `TRTSDT` or `RFSTDTC`-derived `RFSTDT`? The SAP decides; for a safety population analysis it's usually first dose, and `--DY` in SDTM is relative to `RFSTDTC` while `ADY` in ADaM is relative to whatever the SAP says. Second: if `TRTSDT` is missing (screen failure), `ADY` is missing — and any window derived from `ADY` silently drops those records.

Q15. Derive ABLFL. Why isn't there a derive_var_ablfl()?

SHORT ANSWER. Because the baseline definition is a SAP decision, not a standard. You derive it with `derive_var_extreme_flag()` wrapped in `restrict_derivation()`: restrict to records eligible to be baseline, then flag the last (or first) one per subject-parameter-basetype.

FOR THE SAS PROGRAMMER. Same logic as your `%baseline` macro, but the eligibility rule and the "which one wins" rule are two separate, visible arguments instead of one macro parameter. In SAS: sort by USUBJID PARAMCD ADT, `where` on eligibility, `if last.paramcd then ablfl='Y'`. Identical, just declared rather than procedural.

DETAIL. The pieces: `filter` in `restrict_derivation` is eligibility — typically `!is.na(AVAL) & ADT <= TRTSDT & !is.na(BASETYPE)`. `by_vars` includes `BASETYPE` because a crossover or multi-period study has more than one baseline per parameter, and `derive_basetype_records()` is what creates `BASETYPE` (it duplicates records into the relevant analysis periods). `order` is the tie-break — `exprs(ADT, ATPTN, VISITNUM)` — and `mode = "last"` takes the latest pre-dose record. `flag_all = TRUE` is available if you

genuinely want every tied record flagged; the default `FALSE` is what you almost always want, and admiral will error on unresolved ties depending on `check_type`.

```
library(admiral)
library(dplyr)

advs <- restrict_derivation(
  advs,
  derivation = derive_var_extreme_flag,
  args = params(
    by_vars = exprs(STUDYID, USUBJID, BASETYPE, PARAMCD),
    order   = exprs(ADT, ATPTN, VISITNUM),
    new_var = ABLFL,
    mode    = "last"
  ),
  filter = (!is.na(AVAL) & ADT <= TRTSDT & !is.na(BASETYPE))
)
```

INTERVIEW TRAP. "What if two records tie on date and timepoint?" You must break the tie deterministically or `derive_var_extreme_flag()` will complain — add `VISITNUM` or `--SEQ` to `order`. And the deeper trap: if a subject has no pre-treatment record, they get no `ABLFL = "Y"`, hence `BASE` is missing, hence `CHG` is missing, hence they vanish from a change-from-baseline table. That's correct behaviour, but it must be a stated SAP decision, not an accident.

Q16. Derive BASE, CHG and PCHG.

SHORT ANSWER. `derive_var_base()` copies the flagged baseline value onto every record in the by-group; `derive_var_chg()` and `derive_var_pchg()` then take no arguments at all — they compute `AVAL - BASE` and `100 * (AVAL - BASE) / BASE` from the standard ADaM variable names.

FOR THE SAS PROGRAMMER. The zero-argument functions feel odd at first. They're relying on ADaM's own naming contract: if the variables are called `AVAL` and `BASE`, there is exactly one right answer, so there's nothing to parameterise. That's admiral's "simplicity" principle in action — arguments shape how, never what.

DETAIL. `derive_var_base(by_vars = , source_var = AVAL, new_var = BASE, filter = )` — the `filter` argument defaults to selecting the `ABLFL == "Y"` record, and you override it when your baseline flag is named something else or you want `BASEC` from `AVALC`. Include `BASETYPE` in `by_vars` whenever it exists. `derive_var_pchg()` returns `NA` when `BASE` is zero, which is the correct ADaM behaviour and a difference from a naive SAS division that would produce a missing-with-a-note or an infinity depending on how you wrote it.

```
library(admiral)
library(dplyr)

advs <- advs %>%
  derive_var_base(
    by_vars    = exprs(STUDYID, USUBJID, PARAMCD, BASETYPE),
    source_var = AVAL,
    new_var    = BASE
  ) %>%
  derive_var_base(
    by_vars    = exprs(STUDYID, USUBJID, PARAMCD, BASETYPE),
    source_var = AVALC,
    new_var    = BASEC
  ) %>%
  derive_var_chg() %>%
  derive_var_pchg()
```

INTERVIEW TRAP. "Should CHG be populated on the baseline record itself?" ADaM says `CHG` at baseline is zero, and admiral will compute it as zero — but many SAPs blank it. If your spec says blank, you `mutate()` it after. The trap is that a reviewer comparing your table to your dataset will spot the inconsistency, so decide once and document it in the define.

Q17. Derive ANL01FL — how do you pick one record per window?

SHORT ANSWER. There's no `derive_var_anl01fl()` because ANL01FL is purely a SAP construct. You use `derive_var_extreme_flag()` with `by_vars` set to the analysis by-group (subject, parameter, visit) and `order` encoding the selection rule — most commonly "the record closest to the target day, ties broken by the later one."

FOR THE SAS PROGRAMMER. Identical to your `%anl_flag` macro: sort by the tie-break key, `if last.avisit then anl01fl='Y'`. The R version makes the tie-break explicit in `order`, which is the part your macro documentation probably left implicit.

DETAIL. The two common rules. Last record in the window: `order = exprs(ADT, ATPTN), mode = "last"`. Closest to target: create a temporary distance variable and order on it — `mutate(tmp_diff = abs(ADY - AVISITTGT))` then `order = exprs(tmp_diff, desc(ADT))`, `mode = "first"`, then drop the temp variable. admiral will not invent the rule for you, and that is the point: two statisticians will specify this differently and the code has to show which one you implemented. Use `restrict_derivation()` when the flag should only be evaluated within a subset (e.g. post-baseline only), so pre-baseline records keep a blank flag rather than being deleted.

```
library(admiral)
library(dplyr)

advs <- advs %>%
  mutate(tmp_diff = abs(ADY - AVISITTGT)) %>%
  restrict_derivation(
    derivation = derive_var_extreme_flag,
    args = params(
      by_vars = exprs(STUDYID, USUBJID, PARAMCD, AVISIT),
      order   = exprs(tmp_diff, desc(ADT), VISITNUM),
      new_var = ANL01FL,
      mode    = "first"
    ),
    filter = !is.na(AVAL) & !is.na(AVISIT)
  ) %>%
  select(-tmp_diff)
```

INTERVIEW TRAP. "What's the difference between ANL01FL and ABLFL?" Both are `derive_var_extreme_flag()` calls; the difference is the by-group and the eligibility filter. And the follow-up that catches people: ANL01FL must be defined so that exactly one record per analysis by-group is flagged, otherwise your n counts double — so always add a QC step counting flagged records per by-group. Note `derive_var_extreme_flag()` has `flag_all = FALSE` by default precisely to enforce this.

Q18. How do you do AVISIT windowing in admiral?

SHORT ANSWER. admiral does not have a visit-windowing function, and that's a deliberate gap — windowing is entirely protocol-specific. You do it with a windowing metadata table joined on study day, either with `derive_vars_joined()` against a window dataset or with `derive_vars_cat()` / `case_when()` for simple rules. For multi-period designs, `create_period_dataset()` and `derive_vars_period()` handle the `APxxSDT` / `APxxEDT` period variables.

FOR THE SAS PROGRAMMER. This is exactly your visit window lookup dataset joined on `ADY between wstart and wend`. Nothing conceptually new — but be ready for the interviewer to be surprised admiral doesn't do it, because a lot of people assume it does.

DETAIL. The robust pattern is a window metadata data frame (`AVISIT`, `AVISITN`, `AWLO`, `AWHI`, `AWTARGET`) joined with `derive_vars_joined()` using `filter_join = ADY >= AWLO & ADY <= AWHI`. That keeps the windows as data — reviewable, spec-driven, and testable — rather than a 40-branch `case_when()`. For simple named-visit mapping (SCREENING/UNSCHEDULED to missing, "WEEK 4" to `AVISITN = 4`) `derive_vars_cat(definition = )` with a lookup of conditions is cleaner than nested `if_else`. The period machinery is separate: `create_period_dataset()` turns ADSL period variables into a long dataset, and `derive_vars_period()` goes the other way.

```
library(admiral)
library(dplyr)

windows <- tibble::tribble(
  ~AVISIT,     ~AVISITN, ~AWLO, ~AWHI, ~AWTARGET,
  "Baseline",         0,   -28,     1,         1,
  "Week 4",           4,     2,    42,        29,
  "Week 12",         12,    43,   112,        85
)

advs <- derive_vars_joined(
  advs,
  dataset_add = windows,
  by_vars     = NULL,
  new_vars    = exprs(AVISIT, AVISITN, AWTARGET),
  join_vars   = exprs(AWLO, AWHI),
  join_type   = "all",
  filter_join = ADY >= AWLO & ADY <= AWHI,
  mode        = "first",
  order       = exprs(AWLO)
)
```

INTERVIEW TRAP. "What about a record that falls in no window, or in two?" Overlapping windows are a spec bug and your code should surface it, not silently take the first — count matches before collapsing. Records outside all windows get missing `AVISIT` and must then be excluded by `ANL01FL`, not deleted, because the ADaM principle is that you keep the record and flag it. Deleting rows to make a table work is the classic ADaM traceability violation.

Q19. Derive TRTEMFL. What does admiral give you beyond a date comparison?

SHORT ANSWER. `derive_var_trtemfl()` flags treatment-emergent AEs from the onset date versus treatment start and end, with an `end_window` for the post-treatment period. Beyond the simple date logic it handles worsening — an AE that started before treatment but got more severe on treatment — via the `initial_intensity` and `intensity` arguments, and it handles missing onset dates conservatively.

FOR THE SAS PROGRAMMER. Your `%trtemfl` macro almost certainly implemented the date comparison and left worsening to a separate ad hoc step. The worsening logic is where hand-written implementations differ between sponsors and where a validator earns their keep.

DETAIL. Verified arguments: `dataset`, `new_var`, `start_date`, `end_date`, `trt_start_date`, `trt_end_date`, `end_window`, `ignore_time_for_trt_end`, `initial_intensity`, `intensity`, `group_var`, `subject_keys`. `end_window = 30` gives the standard "within 30 days of last dose" rule. `group_var` matters for AEs recorded as multiple records for one event (`AEGRPID`) — the flag then applies at the event level, not the record level. `ignore_time_for_trt_end` controls whether an AE starting on the last dosing day but at an earlier time counts. Note the imputation interaction: if `ASTDTM` was imputed, the flag is only as good as the imputation, which is precisely why you cap the imputation with `min_dates = exprs(TRTSDTM)`.

```
library(admiral)
library(dplyr)

adae <- adae %>%
  derive_var_trtemfl(
    new_var        = TRTEMFL,
    start_date     = ASTDTM,
    end_date       = AENDTM,
    trt_start_date = TRTSDTM,
    trt_end_date   = TRTEDTM,
    end_window     = 30
  ) %>%
  derive_var_ontrtfl(
    new_var         = ONTRTFL,
    start_date      = ASTDT,
    ref_start_date  = TRTSDT,
    ref_end_date    = TRTEDT,
    ref_end_window  = 30
  )
```

INTERVIEW TRAP. "What if the AE onset date is completely missing?" The conservative and conventional answer is to treat it as treatment-emergent — you cannot rule it out — which you achieve by imputing the partial/missing `AESTDTC` to the earliest date consistent with what's known, floored at treatment start. If you answer "exclude it," expect a challenge, because excluding understates the safety profile and no regulator will accept that as a default.

Q20. Build ADTTE. Walk me through censoring.

SHORT ANSWER. You describe the events and the censorings as objects — `event_source()` and `censor_source()` — and hand them to `derive_param_tte()`. admiral picks the earliest event date if any event occurred, otherwise the latest censoring date, sets `CNSR` from the source object (0 for events by CDISC convention, positive integers for censorings), sets `STARTDT` from `start_date`, and carries the imputation flags across automatically.

FOR THE SAS PROGRAMMER. Your `%tte` macro took a list of event datasets and a censoring hierarchy. Same idea, but the hierarchy becomes explicit data: each source carries its own `filter`, `date`, `censor` value and a `set_values_to` block supplying `EVNTDESC`, `SRCDOM`, `SRCVAR`, `SRCSEQ`. That last part is the traceability payload — it tells the reviewer which record in which domain produced this subject's analysis date.

DETAIL. `event_source(dataset_name, filter, date, set_values_to, order)`. `censor_source(dataset_name, filter, date, censor, set_values_to, order, consider_end_dates)`. `derive_param_tte(dataset, dataset_adsl, source_datasets, by_vars, start_date, end_dates, event_conditions, censor_conditions, event_type, create_datetime, set_values_to, subject_keys, check_type)`. `source_datasets` is a named list whose names match each source's `dataset_name`. Rules admiral enforces: `CNSR = 0` for events, default `1` for censorings; `ADT` = earliest event date if an event exists, else latest censoring date; if `start_date` has associated `--DTF`/`--TMF` variables, `STARTDTF`/`STARTTMF` are populated automatically. `lastalive_censor` is a ready-made censoring source for last-known-alive. Multiple parameters are built by calling `derive_param_tte()` once per `PARAMCD` and stacking.

```
library(admiral)
library(dplyr)

death <- event_source(
  dataset_name  = "adsl",
  filter        = DTHFL == "Y",
  date          = DTHDT,
  set_values_to = exprs(EVNTDESC = "DEATH", SRCDOM = "ADSL", SRCVAR = "DTHDT")
)

lstalv <- censor_source(
  dataset_name  = "adsl",
  date          = LSTALVDT,
  censor        = 1,
  set_values_to = exprs(EVNTDESC = "LAST KNOWN ALIVE DATE", SRCDOM = "ADSL", SRCVAR = "LSTALVDT")
)

adtte <- derive_param_tte(
  dataset_adsl      = adsl,
  source_datasets   = list(adsl = adsl),
  start_date        = TRTSDT,
  event_conditions  = list(death),
  censor_conditions = list(lstalv),
  set_values_to     = exprs(PARAMCD = "OS", PARAM = "Overall Survival")
)
```

INTERVIEW TRAP. "What if the censoring date is before the start date?" You get a negative `AVAL`, and admiral will not stop you — it's your check. Also expect: "why is CNSR = 0 the event?" Because the ADaM Basic Data Structure for Time-to-Event says so, and it's the opposite of the R `survival` package's convention where the status indicator is 1 for an event. When you hand `AVAL` / `CNSR` to `survival::Surv()`, you write `Surv(AVAL, 1 - CNSR)`. Getting that backwards inverts your entire Kaplan-Meier curve, and it is the single most common R-side ADTTE bug.

Q21. How do you add derived parameters to a BDS dataset?

SHORT ANSWER. `derive_param_computed()` for a parameter computed from other parameters at the same timepoint (BMI from height and weight, mean arterial pressure from systolic and diastolic). `derive_summary_records()` for a parameter that summarises records (mean over a visit). `derive_extreme_records()` for "worst post-baseline value" style parameters. All three add rows, not columns.

FOR THE SAS PROGRAMMER. These are your `output;` statements. The gain is that the by-group logic, the constant-variable handling and the "which parameters must be present" contract are arguments rather than assumptions — `derive_param_computed()` will simply not produce a record if a required input parameter is missing, instead of producing a silently wrong one.

DETAIL. `derive_param_computed(dataset, dataset_add, by_vars, parameters, set_values_to, filter, constant_by_vars, constant_parameters, keep_nas)`. `parameters` names the input `PARAMCD`s; inside `set_values_to` you refer to them as `AVAL.PARAMCD` — e.g. `AVAL.WEIGHT / (AVAL.HEIGHT/100)^2`. `constant_parameters` handles inputs measured once (height at screening) rather than at every visit, with `constant_by_vars` naming the level at which they're constant. `derive_summary_records(dataset, dataset_add, dataset_ref, by_vars, filter_add, set_values_to, missing_values)` — `dataset_ref` lets you force records to exist even when there's nothing to summarise, which is how you get a row of zeros instead of a missing row.

```r
library(admiral)
library(dplyr)

advs <- derive_param_computed(
  advs,
  by_vars       = exprs(USUBJID, AVISIT, AVISITN, ADT, ADY),
  parameters    = c("WEIGHT", "HEIGHT"),
  set_values_to = exprs(
    AVAL   = AVAL.WEIGHT / (AVAL.HEIGHT / 100)^2,
    PARAMCD = "BMI",
    PARAM   = "Body Mass Index (kg/m^2)"
  )
)

# Mean of triplicate readings within a visit
advs_mean <- derive_summary_records(
  advs,
  dataset_add   = advs,
  by_vars       = exprs(STUDYID, USUBJID, PARAMCD, AVISIT, AVISITN),
  set_values_to = exprs(AVAL = mean(AVAL, na.rm = TRUE), DTYPE = "AVERAGE")
)
```

INTERVIEW TRAP. `DTYPE`. Any record you create rather than observe must carry a `DTYPE` value (AVERAGE, LOCF, MAXIMUM, PHANTOM...) so a reviewer can tell derived rows from source rows, and your analysis flags must then decide whether to include them. Forgetting `DTYPE` is a define.xml and ADRG finding waiting to happen. Related: `derive_locf_records()` creates LOCF rows and you must set `DTYPE = "LOCF"`.

Q22. ANRIND, toxicity grades, criterion flags — what does admiral provide?

SHORT ANSWER. `derive_var_anrind()` derives the reference range indicator from `AVAL` against `ANRLO`/`ANRHI` (and optionally `A1LO`/`A1HI`). `derive_var_atoxgr_dir()` and `derive_var_atoxgr()` derive NCI-CTCAE grades from a supplied criteria metadata table. `derive_vars_crit_flag()` builds the `CRITy`/`CRITyFL` pairs. `derive_var_shift()` builds shift variables like `SHIFT1` from baseline and analysis range indicators.

FOR THE SAS PROGRAMMER. Same derivations you already do; the interesting design choice is that the toxicity criteria are data, not code. You pass a metadata table describing each lab test's grading thresholds and direction, so adding a new CTCAE version is a data change rather than a rewrite.

DETAIL. `derive_var_anrind(dataset, signif_dig, use_a1hia1lo)` — note `signif_dig`, which controls the rounding used when comparing `AVAL` to the limits; this exists because floating-point comparison at the boundary is a real source of PD-vs-QC disagreement. `derive_var_atoxgr_dir(dataset, new_var, tox_description_var, meta_criteria, criteria_direction, abnormal_indicator, high_indicator, low_indicator, get_unit_expr, signif_dig)` grades in one direction; `derive_var_atoxgr()` combines the high and low grade variables into `ATOXGR`. admiral ships a criteria metadata dataset for CTCAE — check the Lab Grading vignette for the version currently supplied, because CTCAE versions are added over time. `derive_vars_crit_flag(dataset, crit_nr, condition, description, values_yn, create_numeric_flag)`.

```
library(admiral)
library(dplyr)

adlb <- adlb %>%
  derive_var_anrind() %>%
  derive_var_shift(
    new_var  = SHIFT1,
    from_var = BNRIND,
    to_var   = ANRIND
  ) %>%
  derive_vars_crit_flag(
    crit_nr     = 1,
    condition   = AVAL > 3 * ANRHI,
    description = "ALT > 3xULN",
    values_yn   = TRUE
  )
```

INTERVIEW TRAP. “Why does `derive_var_anrind()` have a `signif_dig` argument?” Because `AVAL` and `ANRHI` are doubles, and a value that displays as exactly equal to the upper limit may compare as greater. Rounding both to a stated number of significant digits before comparing makes the boundary deterministic and reproducible. If you’ve ever chased a single-cell PD/QC difference on an out-of-range count, this is the answer, and knowing it signals real experience.

Section 4 — Partial dates and imputation

Q23. Explain partial date imputation in admiral. Why do the impute arguments matter so much?

SHORT ANSWER. `derive_vars_dt()` and `derive_vars_dtm()` take `highest_imputation`, and the default is `"n"` — no imputation at all. So a partial `--DTC` gives you `NA` unless you explicitly ask for imputation. Everything after that (`date_imputation`, `time_imputation`, `min_dates`, `max_dates`, `preserve`, `flag_imputation`) tells admiral exactly what rule you chose, and the rule is a SAP decision.

FOR THE SAS PROGRAMMER. The reversal of default is the key point. Most SAS date macros impute by default because they were written to always return something. admiral fails loudly — you get missing values you can see — rather than quietly inventing a date. That is a deliberate safety property, and it is the answer to "why is my ASTDT all missing?"

DETAIL. `highest_imputation` is the highest component admiral is allowed to invent: `"n"` none, `"s"` seconds, `"h"` hours, `"D"` day, `"M"` month, `"Y"` year. All components below the level are also imputed; if a component above the level is missing you get `NA` — so `"2020"` with `highest_imputation = "D"` stays `NA`, because the month is missing and you only authorised day-level imputation. `date_imputation` accepts `"first"`, `"last"`, `"mid"`, or a literal like `"01-01"` / `"06-15"`; prefer the keywords, because a literal `"12-31"` applied to a partial February date produces an invalid date. `"mid"` imputes a missing day to the 15th and a missing day-and-month to 06-30. `preserve` controls whether known components survive: with `"2019---07"` (year known, month missing, day 07) and `preserve = TRUE`, the day 07 is kept. `time_imputation` takes `"first"`, `"last"` or a literal like `"00:00:00"`. `flag_imputation` controls whether `--DTF` / `--TMF` are created; the default derives them, and you should leave it alone.

```
library(admiral)
library(dplyr)

ae <- ae %>%
  derive_vars_dtm(
    dtc                = AESTDTC,
    new_vars_prefix    = "AST",
    highest_imputation = "M",
    date_imputation    = "first",
    time_imputation    = "first",
    min_dates          = exprs(TRTSDTM)
  ) %>%
  derive_vars_dtm(
    dtc                = AEENDTC,
    new_vars_prefix    = "AEN",
    highest_imputation = "M",
    date_imputation    = "last",
    time_imputation    = "last",
    max_dates          = exprs(DTHDT, DCUTDT)
  ) %>%
  derive_vars_dtm_to_dt(source_vars = exprs(ASTDTM, AENDTM))
```

INTERVIEW TRAP. "What do `min_dates` and `max_dates` actually do?" They bound the imputed value, and crucially they only consider dates that fall inside the range the partial `--DTC` already implies — so a non-missing component is never overwritten. Practically: `min_dates = exprs(TRTSDTM)` stops you imputing an AE onset to a date before treatment started (which would make a genuinely treatment-emergent AE look pre-existing), and `max_dates = exprs(DTHDT, DCUTDT)` stops an imputed end date

landing after death or after the data cut. Year-level imputation (`highest_imputation = "Y"`) is only permitted when you supply `min_dates` with `"first"` or `max_dates` with `"last"` — admiral refuses otherwise, because inventing a year unbounded is indefensible.

Q24. What are DTF and TMF, and why can't you skip them?

SHORT ANSWER. `--DTF` and `--TMF` are the ADaM date and time imputation flags — they record which component was imputed (`"D"`, `"M"`, `"Y"` for date; `"S"`, `"M"`, `"H"` for time). ADaM requires that if you impute, you flag. admiral derives them automatically alongside the imputed variable, and exports `compute_dtf()` / `compute_tmf()` if you need them standalone.

FOR THE SAS PROGRAMMER. Same variables you already create. The difference is that in admiral you get them for free and would have to actively switch them off with `flag_imputation = "none"` — the safe default is the one that produces the compliant output.

DETAIL. `derive_vars_dtm(dtc = AESTDTC, new_vars_prefix = "AST", ...)` produces `ASTDTM`, `ASTDTF` and `ASTTMF`. `flag_imputation` accepts `"auto"` (default, derive whichever flags are relevant), `"date"`, `"time"`, `"both"`, `"none"`. The one legitimate reason to use `"none"` is when you're deriving an intermediate variable that never reaches the final dataset. Downstream, `derive_param_tte()` looks for `--DTF` / `--TMF` on the `start_date` and automatically produces `STARTDTF` / `STARTTMF` — traceability propagating without you writing it.

```
library(admiral)

# Vector-level equivalents, if you need the flag without the imputed value
compute_dtf(dtc = "2019-07", dt = as.Date("2019-07-01"))
compute_tmf(dtc = "2019-07-18T15", dtm = as.POSIXct("2019-07-18T15:00:00", tz = "UTC"))
```

INTERVIEW TRAP. A Pinnacle 21 or define.xml reviewer will ask why `ASTDT` is populated where `AESTDTC` was partial and `ASTDTF` is blank. There is no good answer — it means either you imputed without flagging, or the flag got dropped in a later `select()`. Add a check: every record where the source `--DTC` is partial and the derived date is non-missing must have a non-blank `--DTF`.

Section 5 — SDTM in R, and specification-driven programming

Q25. What is sdtm.oak and how does it work?

SHORT ANSWER. sdtm.oak is pharmaverse's EDC-agnostic SDTM automation package. Its model is that SDTM mapping decomposes into a small set of repeatable algorithms — assign a value with or without controlled terminology, hardcode a value with or without CT, apply a condition, build an ISO 8601 date — which you compose per target variable, keeping a link back to the originating raw record.

FOR THE SAS PROGRAMMER. Think of it as formalising what your raw-to-SDTM programs already do, but naming the four or five patterns explicitly instead of writing a bespoke DATA step per domain. The traceability idea is the genuinely new bit: `generate_oak_id_vars()` stamps each raw record with an identifier that survives into the SDTM domain, so you can point at an SDTM row and name the raw row it came from.

DETAIL. The exported API includes `assign_no_ct()`, `assign_ct()`, `hardcode_no_ct()`, `hardcode_ct()`, `condition_add()`, `create_iso8601()`, `assign_datetime()`, `derive_seq()`, `derive_study_day()`, `derive_blfl()`, `generate_oak_id_vars()`, `oak_id_vars()`, `generate_sdtm_supp()`, `read_ct_spec()` and `ct_map()`. Controlled terminology lives in a CT specification read with `read_ct_spec()`, and the `_ct` variants of the assign/hardcode algorithms consult it. `condition_add()` is the conditional-mapping wrapper — "map this only where the raw record satisfies X" — which is how you avoid a nest of `if_else`.

```
#| skip-check
# Illustrative - sdtm.oak is not installed in this environment.
library(sdtm.oak)

vs <- raw_vs %>%
  generate_oak_id_vars(pat_var = "PATNUM", raw_src = "vs_raw") %>%
  assign_no_ct(tgt_var = "VSORRES", raw_var = "VSTESTRESULT", raw_dat = .) %>%
  assign_ct(tgt_var = "VSTESTCD", raw_var = "VSTESTNAME",
            ct_spec = ct, ct_clst = "C66741", raw_dat = .)
# Verify argument names against the current sdtm.oak reference before quoting
# them in an interview - this package is young and its API is still settling.
```

INTERVIEW TRAP. Don't oversell it. sdtm.oak is a young package and its API has moved. The honest framing: "it gives you a vocabulary of mapping algorithms and record-level traceability, which is the right idea; I'd pilot it on one domain before betting a study on it." If you're asked whether you'd use it for a filing right now, saying "not yet, without a pilot" is the mature answer.

Q26. Why is SDTM in R less mature than ADaM in R?

SHORT ANSWER. Because ADaM derivations are standardised and SDTM mapping isn't. ADaM tells you exactly what `ABLFL`, `CHG` and `ADY` mean, so a library of derivations is reusable across sponsors. SDTM tells you what the target looks like but says nothing about your CRF, your EDC export, or your data-collection conventions — so 80% of the work is study-specific mapping decisions that no package can encode.

FOR THE SAS PROGRAMMER. You already know this: your ADaM macros port between studies; your raw-to-SDTM programs mostly don't. The R ecosystem simply reflects that reality honestly.

DETAIL. Concrete gaps beyond the mapping problem itself. Trial design domains (TS, TA, TE, TV, TI, TD) are mostly hand-built from the protocol, and there's no R tooling worth the name. SUPPQUAL and RELREC are mechanical but fiddly — sdtm.oak has `generate_sdtm_supp()` and metatools has `make_supp_qual()` / `build_qnam()` / `combine_supp()`, which covers a lot of it. Conformance checking is still dominated by Pinnacle 21; on the open-source side `sdtmchecks` implements a library of data-quality checks but is not a conformance validator and does not replace P21 for a submission. And the define.xml for SDTM has the same generation gap as ADaM.

```
library(metatools)
library(dplyr)

# The SUPPQUAL round trip, which R does handle well:
# combine_supp() merges a SUPP-- dataset back onto its parent as columns
ae_full <- combine_supp(ae, suppae)

# and the reverse, building SUPP-- from flagged columns:
# make_supp_qual(dataset, metacore, dataset_name)
```

INTERVIEW TRAP. If asked "would you replace your SDTM programming with R?", the differentiating answer separates the two halves: the mechanical half — SEQ derivation, study day, baseline flags, SUPPQUAL assembly, ISO 8601 construction, CT application — automates well and R handles it fine. The judgement half — which CRF field maps to which variable, how to handle the site that recorded units in a comment field — does not automate, in any language. Anyone selling you full SDTM automation is selling you the easy half.

Q27. What is metacore and what problem does it solve?

SHORT ANSWER. metacore turns a specification — an Excel spec or a define.xml — into an immutable R object holding dataset metadata, variable metadata, codelists, derivations and keys. Once the spec is an object, code can consult it rather than duplicating it, which is how you stop the spec and the code drifting apart.

FOR THE SAS PROGRAMMER. This is the difference between a spec you read and a spec you execute. In a SAS shop the spec workbook defines the label, length, type and order of every variable, and then a human retypes all of that into an `attrib` statement. Every retyping is a chance for the code and the define to disagree. metacore removes the retyping.

DETAIL. Entry points: `spec_to_metacore(path, quiet, where_sep_sheet, verbose)` for a Pinnacle-21-style Excel spec, and `define_to_metacore(path, quiet, verbose)` for a define.xml. Both return a metacore object covering all datasets; `select_dataset(.data, dataset)` narrows it to one, at which point the object carries that dataset's variables, labels, types, lengths, order, keys, codelists and derivation text. Accessors include `get_control_term(metacode, variable, dataset)` for a codelist and `get_keys(metacode, dataset)` for the sort keys. The object is validated on construction — `check_inconsistent_types()`, `check_inconsistent_labels()` and `check_inconsistent_formats()` will tell you when the same variable is specified differently in two datasets, which is a define.xml finding you'd otherwise discover after submission.

```
library(metacore)

spec_path <- metacore_example("p21_mock.xlsx")
mc <- spec_to_metacore(spec_path, quiet = TRUE)

adsl_spec <- select_dataset(mc, "ADSL")
get_keys(mc, "ADSL")
get_control_term(mc, variable = "SEX", dataset = "ADSL")
```

INTERVIEW TRAP. "What if the spec is wrong?" metacore will faithfully apply a wrong spec — it is not a validator of your content, only of internal consistency. The value is that the error now appears in exactly one place and the fix is one cell in the workbook, not a hunt through fifteen programs. Also be ready for: metacore reads a Pinnacle-21-format Excel spec; a bespoke company spec workbook needs a mapping layer first, and that mapping layer is real work.

Q28. What does metatools do that admiral doesn't?

SHORT ANSWER. metatools applies the metacore spec to the data. admiral derives; metatools enforces. It builds predecessor variables straight from the spec, creates variables from codelists, checks that the dataset has exactly the variables the spec says, checks values against controlled terminology, drops unspecified variables, applies labels, orders columns and sorts by key.

FOR THE SAS PROGRAMMER. This is the `attrib` / `retain` / `keep` block plus your compliance-check macro, driven from the spec instead of hand-written. `check_variables()` is "did I create everything the spec asked for and nothing extra"; `check_ct_data()` is your controlled-terminology check macro.

DETAIL. Key functions and their spec-driven signatures: `build_from_derived(metacore, ds_list, dataset_name, predecessor_only, keep, verbose)` reads the spec's derivation column and pulls predecessor variables from the named SDTM datasets. `create_var_from_codelist(data, metacore, input_var, out_var, codelist, decode_to_code, strict)` builds a decode from a code (or vice versa) using the spec's codelist rather than a hard-coded lookup. `create_cat_var(data, metacore, ref_var, grp_var, num_grp_var, ...)` builds grouping variables from a spec codelist. `check_variables(data, metacore, dataset_name, strict)` errors on missing or extra variables. `check_ct_data(data, metacore, na_acceptable, omit_vars, verbose)` checks values against controlled terminology, with `get_bad_ct()` returning the offenders. `drop_unspec_vars()`, `order_cols()`, `sort_by_key()`, `set_variable_labels()` finish the dataset.

```
library(metacore)
library(metatools)
library(dplyr)

adsl_final <- adsl %>%
  check_variables(mc, dataset_name = "ADSL") %>%   # errors on missing/extra vars
  check_ct_data(mc, na_acceptable = TRUE) %>%      # CT compliance
  order_cols(mc, dataset_name = "ADSL") %>%
  sort_by_key(mc, dataset_name = "ADSL") %>%
  set_variable_labels(mc, dataset_name = "ADSL")
```

INTERVIEW TRAP. The pipeline order. `drop_unspec_vars()` must come before `check_variables()` if you have working variables, or `check_variables()` fails on the extras. And `order_cols()` before `xportr_order()` is redundant — pick one source of truth. The deeper point an interviewer is probing: these checks should error and stop the run, not print a warning nobody reads. In R that's a real

decision, because `strict = FALSE` turns errors into messages, and a pipeline that warns is a pipeline that ships bad data.

Q29. How do you handle controlled terminology checks in R?

SHORT ANSWER. Two layers. Data-level: `check_ct_data()` / `check_ct_col()` / `get_bad_ct()` from metatools compare your values against the codelists in the metacore object. Submission-level: Pinnacle 21 remains the authority for conformance, because it checks against the published CDISC CT versions and the FDA validator rules, not just your spec.

FOR THE SAS PROGRAMMER. Same two layers you already have — the in-program check and the P21 run. What changes is that the in-program check reads the codelist from the spec object instead of a hard-coded `%let` list, so updating a CT version is a spec change.

DETAIL. `check_ct_data(data, metacore, na_acceptable, omit_vars, verbose)` checks every variable with a codelist in the spec; `na_acceptable` controls whether missing counts as a violation, and `omit_vars` skips variables you know are free text. `get_bad_ct(data, metacore, var, na_acceptable)` returns the failing values so you can look at them rather than guess. For extensible codelists you have to be careful: a value not in the codelist may be legitimate, so `check_ct_data()` flagging it is information, not necessarily an error — that judgement stays with you. There is also a `meddra.read` package for MedDRA dictionary loading, but MedDRA licensing means most sponsors handle that internally.

```
library(metacore)
library(metatools)

# Which values fail CT, and for which variable
bad <- get_bad_ct(adsl, mc, var = "ETHNIC", na_acceptable = TRUE)
bad

# Whole-dataset check; returns the data invisibly so it pipes
# adsl <- check_ct_data(adsl, mc, na_acceptable = TRUE)
```

INTERVIEW TRAP. "Does this replace Pinnacle 21?" No, and saying it does is a red flag. metatools checks your data against your spec. P21 checks your data, your define and your spec against the published CDISC standards and the FDA/PMDA validator rule sets, and produces the validation report that goes in the submission package. The R checks are the fast inner loop; P21 is the gate. Worth knowing: in Pilot 5 the team could not validate Dataset-JSON v1.1 in Pinnacle 21 at all and fell back to validating the XPT versions, documenting the gap in the ADRG — which tells you exactly how central P21 still is.

Section 6 — TLFs in R

Q30. Explain the rtables layout paradigm.

SHORT ANSWER. rtables separates the layout from the data. You declare a table shell — how columns split, how rows split, what analysis runs in each cell — as an object, before any data is involved. Then `build_table(lyt, df)` applies it. The result is a table object with addressable row and column paths, not a data frame, so you can sort it, prune it, footnote a specific cell, and paginate it.

FOR THE SAS PROGRAMMER. The nearest analogue is writing the shell first and then pointing a proc at it — but no SAS proc really works this way. `PROC REPORT` interleaves the data step and the display step; rtables makes the display step a first-class object you can inspect and manipulate. The payoff is pagination and referential footnotes that actually work, and the cost is a genuinely new mental model.

DETAIL. The vocabulary: `basic_table()` starts an empty layout and takes `title`, `subtitles`, `main_footer`, `prov_footer`, and `show_colcounts`. `split_cols_by(var)` creates one column per level, and splits nest. `split_rows_by(var)` does the same for rows; `summarize_row_groups()` turns a plain group label row into a content row (typically n and percent). `analyze(vars, afun, format)` is the leaf — the function that fills the cells. `add_overall_col(label)` adds a total column. `append_topleft()` puts text in the top-left stub. `build_table(lyt, df, alt_counts_df)` runs it — `alt_counts_df` is how you get denominators from a different population (ADSL) than the analysis data (ADAE), which is the classic "N in the header comes from the population, n in the body from the AEs" problem. Afterwards `prune_table()`, `trim_rows()` and `sort_at_path()` operate on the built object.

```
library(rtables)
library(dplyr)

lyt <- basic_table(
    title       = "Table 14.1.1  Demographics - Safety Population",
    subtitles   = "Protocol XYZ-123",
    main_footer = "Percentages are based on the number of subjects in the safety population.",
    prov_footer = "Program: t_dm.R   Output: t_dm.rtf",
    show_colcounts = TRUE
  ) %>%
  split_cols_by("ARM") %>%
  add_overall_col("All Patients") %>%
  split_rows_by("SEX", split_label = "Sex", label_pos = "topleft") %>%
  summarize_row_groups() %>%
  analyze("AGE", afun = mean, format = "xx.x")

tbl <- build_table(lyt, ex_adsl)
tbl
```

INTERVIEW TRAP. "How do you get the header N from ADSL when the table is built on ADAE?" `alt_counts_df = adsl` in `build_table()`. People miss this and end up with column counts equal to the number of AE records. The related trap is percentages: `summarize_row_groups()` defaults to a percentage relative to the column's data, so a denominator that should be the safety population needs `alt_counts_df` set, not a post-hoc fix.

Q31. What is Tplyr and who is it for?

SHORT ANSWER. Tplyr is a table-building package explicitly designed for SAS programmers. You declare layers — a count layer, a descriptive-statistics layer, a shift layer — attached to a table, set format strings that look like SAS formats, and `build()` returns a tidy data frame you then render with whatever you like. Its other selling point is metadata: you can trace a rendered cell back to the exact source rows.

FOR THE SAS PROGRAMMER. Layers map almost one-to-one onto what you already do: `group_count()` is a `PROC FREQ` block, `group_desc()` is a `PROC MEANS` block, `group_shift()` is a shift table. `set_format_strings(f_str("xx (xx.x%)", n, pct))` is a picture format. If your team is transitioning from SAS, Tplyr is the shortest bridge in the ecosystem.

DETAIL. `tplyr_table(data, treat_var, where = )` creates the table; `add_layer()` attaches a layer; `group_count(target)`, `group_desc(target)`, `group_shift(vars(row = , column = ), by = )` are the three layer types; `set_format_strings()` with `f_str()` controls the display; `set_distinct_by(USUBJID)` switches counting from records to distinct subjects (the "subjects with at least one AE" problem); `add_total_row()`, `set_pop_data()` for a separate population dataset, `set_custom_summaries()` for your own statistics. Everything is lazy — building the object only assembles metadata, and `build()` triggers the computation. `get_numeric_data()` returns the pre-formatted numbers, which is exactly what a validator wants to compare against.

```
#| skip-check
# Illustrative - Tplyr is not installed in this environment.
library(Tplyr)

t <- tplyr_table(adsl, TRT01P, where = SAFFL == "Y") %>%
  add_layer(
    group_desc(AGE, by = "Age (years)") %>%
      set_format_strings(
        "n"         = f_str("xx", n),
        "Mean (SD)" = f_str("xx.x (xx.xx)", mean, sd),
        "Median"    = f_str("xx.x", median),
        "Min, Max"  = f_str("xx, xx", min, max)
      )
  ) %>%
  add_layer(
    group_count(AGEGR1, by = "Age category, n (%)") %>%
      set_format_strings(f_str("xx (xx.x%)", n, pct))
  )

out <- build(t)          # a tidy data frame; render with gt/flextable/r2rtf
```

INTERVIEW TRAP. "So Tplyr produces the output?" No — it produces a data frame. Rendering, pagination, page headers and RTF encoding are still yours, typically via gt, flextable, huxtable+pharmaRTF, or r2rtf. That boundary is the single most important thing to understand about Tplyr, and it's also the reason a team sometimes picks rtables instead: rtables owns the whole path to the page.

Q32. rtables vs Tplyr vs gt/flextable — when do you use which?

SHORT ANSWER. rtables/tern when you need a structured table object with real pagination, referential footnotes and programmatic QC — and you're prepared for the learning curve. Tplyr when your team thinks in PROC FREQ and PROC MEANS and you want fast, traceable summary data you'll render

separately. gt and flextable are renderers, not summarisers — they turn a data frame into a beautiful table and know nothing about clinical pagination.

FOR THE SAS PROGRAMMER. Roughly: rtables ≈ a report-writing framework, Tplyr ≈ your summary-stats macros, gt/flextable ≈ ODS destinations. You need one from the first two and one from the third — or rtables plus its `formatters` exporters, which covers both.

DETAIL. A fuller map of the space. **tern** sits on rtables and supplies pre-built clinical analysis functions (`analyze_vars()`, `count_occurrences()`, `summarize_ancova()`, `count_abnormal()` and many more) so you're not writing `afun`s from scratch — for standard safety and efficacy tables tern is where the leverage is. **rlistings** does listings in the same paradigm. **formatters** does the rendering and pagination for both. **tfrmt** takes a different angle: it separates the format metadata from the data so you can define a display shell once and reuse it. **tidytlg** (GSK) is a tidyverse-flavoured table builder. **gtsummary** is excellent for exploratory and publication tables but is not built for regulatory display control. **r2rtf** (Merck) is the RTF workhorse. Pick one summarisation paradigm and one rendering path per study, and write it into the programming plan.

```
library(rtables)
library(tern)
library(dplyr)

# tern supplies the analysis functions so you don't hand-write afuns
lyt <- basic_table(show_colcounts = TRUE) %>%
  split_cols_by("ARM") %>%
  tern::analyze_vars("AGE", .stats = c("n", "mean_sd", "median", "range"))

tbl <- build_table(lyt, ex_adsl)
tbl
```

INTERVIEW TRAP. The honest answer to "which is best" is "which one does your team already have programs in." Switching paradigms mid-portfolio costs more than any of them saves. If pushed for a personal view, a defensible one is: rtables plus tern for tables and listings because pagination and footnotes are solved rather than hand-rolled; Tplyr when the team is mid-migration from SAS and needs velocity. Say the trade-off, don't declare a winner.

Q33. How do you produce an RTF or PDF a sponsor will actually accept?

SHORT ANSWER. Two viable paths. Path one: rtables plus `formatters::export_as_rtf()` / `export_as_txt()` / `export_as_pdf()`, with `paginate_table()` controlling lines and characters per page. Path two: r2rtf, which is purpose-built for clinical RTF — `rtf_page()`, `rtf_title()`, `rtf_colheader()`, `rtf_body()`, `rtf_footnote()`, `rtf_source()`, then `rtf_encode()` and `write_rtf()`.

FOR THE SAS PROGRAMMER. This replaces `ODS RTF` plus `PROC REPORT` plus the title/footnote statements. The things you'll immediately look for — repeat the column header on every page, keep the footnote block with the table, "Page X of Y", a source line with the program path, landscape orientation, a monospaced font so the legacy outputs line up — are all supported, but you set them explicitly.

DETAIL. For rtables: `paginate_table(tt, page_type, font_family, font_size, lineheight, landscape, pg_width, pg_height, margins, lpp, cpp, min_siblings)` — `lpp` is lines per page, `cpp` characters per page, and `min_siblings` is orphan control (don't strand one row of a group on its own page). `split_rows_by(var, page_by = TRUE)` forces a page break per level with an automatic page title. Titles and footers come from `basic_table()`. For r2rtf: `rtf_page(tbl, orientation, width, height, margin,`

`nrow, ...)` where `nrow` is rows per page; `rtf_title(tbl, title, subtitle, ...)`; `rtf_colheader(tbl, colheader, col_rel_width)`; `rtf_body(tbl, col_rel_width, ...)`; `rtf_footnote()`; `rtf_source()`; `rtf_encode(tbl, doc_type, page_title, page_footnote, page_source)`; `write_rtf(rtf, file)`. r2rtf can also convert RTF to PDF or DOCX via LibreOffice.

```
library(rtables)
library(formatters)

tbl <- build_table(lyt, ex_adsl)

export_as_rtf(
  tbl,
  file        = "t_dm.rtf",
  landscape   = TRUE,
  font_family = "Courier",
  font_size   = 8,
  margins     = c(1, 1, 1, 1)
)

export_as_txt(tbl, file = "t_dm.txt", paginate = TRUE, lpp = 55, cpp = 130)
```

```
library(r2rtf)
library(dplyr)

head(iris, 20) %>%
  rtf_page(orientation = "landscape", nrow = 10) %>%
  rtf_title(title = "Table 14.1.1  Demographics",
            subtitle = "Safety Population") %>%
  rtf_colheader(colheader = "Sepal L | Sepal W | Petal L | Petal W | Species",
                col_rel_width = c(2, 2, 2, 2, 3)) %>%
  rtf_body(col_rel_width = c(2, 2, 2, 2, 3)) %>%
  rtf_footnote(footnote = "Percentages based on non-missing observations.") %>%
  rtf_source(source = "Program: t_dm.R") %>%
  rtf_encode() %>%
  write_rtf("t_dm.rtf")
```

INTERVIEW TRAP. "Can you produce a PDF?" `export_as_pdf()` produces a monospaced, text-rendered PDF — effectively the .lst in a PDF wrapper. That is exactly right for a submission-style listing and exactly wrong if someone expects a typeset, proportional-font document. Know which one your sponsor wants before you promise it. Second trap: RTF page-break control depends on knowing rows per page in advance; a table whose row heights vary (wrapped long AE terms) will paginate badly unless you set widths and test with the longest real term, not with the example data.

Q34. How do you QC a table in R — and how is that different from SAS?

SHORT ANSWER. The big advantage of rtables is that the built table is an object with addressable paths, so a QC program can pull a specific cell by path and compare it to an independently computed value, rather than diffing text output. With Tplyr, `get_numeric_data()` gives you the pre-formatting numbers for the same purpose. Both beat comparing .lst files.

FOR THE SAS PROGRAMMER. In SAS the QC comparison is usually either a `PROC COMPARE` of the summary dataset or a visual/diff comparison of the listing text. Comparing text means you're comparing display rounding as well as arithmetic, and a mismatch tells you nothing about which of the two it was. Comparing the numeric object separates the two failure modes.

DETAIL. Practical pattern: the QC programmer independently derives the numbers into a plain data frame, and the comparison is `diffdf(qc_numbers, prod_numbers, keys = c("PARAMCD", "AVISIT", "TRT01A", "STAT"))`. `diffdf` reports differences by key with a tolerance argument, which is the R equivalent of `PROC COMPARE ... CRITERION=`. For the rendered table, compare the formatted strings only as a second, separate check — that one is testing your format strings, not your statistics. And keep the raw output of both runs so a third person can adjudicate; the tlf-debugger discipline of recomputing independently from ADaM rather than comparing the two outputs to each other applies exactly the same in R.

```
library(diffdf)
library(dplyr)

# PD vs QC on the numeric layer, not the rendered text
res <- diffdf(
  base    = qc_summary,
  compare = prod_summary,
  keys    = c("PARAMCD", "AVISIT", "TRTA", "STAT"),
  tolerance = 1e-8
)
res
```

INTERVIEW TRAP. "The two outputs differ in one cell — what do you do first?" Do not start by reading either program. Recompute that single cell independently from the ADaM data, decide which side is right, and only then read the wrong program. Comparing PD to QC directly tells you they disagree, never who is correct — and a surprising fraction of the time both are wrong in different ways.

Section 7 — Submission: XPT, define, pilots, traceability, dual programming

Q35. What does xportr do and why does it exist?

SHORT ANSWER. xportr takes a data frame plus the spec and produces a compliant SAS Version 5 transport file: correct types, lengths, labels, formats and column order, with a dataset label. It exists because `haven::write_xpt()` writes a valid XPT but knows nothing about your spec — it won't set variable labels from metadata, won't enforce lengths, won't order columns, and won't tell you when you've breached a V5 limit.

FOR THE SAS PROGRAMMER. This is the `attrib` statement plus `PROC COPY` to a `LIBNAME ... XPORT`. In SAS, the attributes ride with the dataset and the transport engine enforces the V5 rules for you. In R, a data frame has no labels, no lengths and no formats — those are attributes xportr attaches — so this step is genuinely necessary rather than ceremonial.

DETAIL. Six main functions, applied in a pipeline: `xportr_type()` coerces to the only two XPT types (character and numeric/double); `xportr_length()` applies the spec widths; `xportr_label()` applies variable labels; `xportr_order()` reorders columns to spec order (variables not in the spec are pushed to the end with a message); `xportr_format()` applies SAS formats such as `DATE9.` or `DATETIME20.`; `xportr_write()` writes the file. Plus `xportr_df_label()` for the dataset label, `xportr_metadata()` to set the spec and domain once so the downstream calls don't repeat them, `xportr_split()` for splitting oversized datasets, and `xpt_validate()` which runs the compliance checks. `xportr_write(.df, path, max_size_gb = NULL, metadata = NULL, domain = NULL, strict_checks = FALSE)` — note `strict_checks = FALSE` by default warns and writes anyway; for a submission run you want `TRUE` so a failure aborts.

```
library(xportr)
library(readxl)
library(dplyr)

var_spec <- read_xlsx(
    system.file("specs", "ADaM_spec.xlsx", package = "xportr"),
    sheet = "Variables"
  ) %>%
  rename(type = "Data Type") %>%
  rename_with(tolower)

adsl %>%
  xportr_metadata(var_spec, domain = "ADSL") %>%
  xportr_type() %>%
  xportr_length() %>%
  xportr_label() %>%
  xportr_order() %>%
  xportr_format() %>%
  xportr_df_label(var_spec) %>%
  xportr_write("adsl.xpt", strict_checks = TRUE)
```

INTERVIEW TRAP. "Why not just `haven::write_xpt()`?" Because it will happily write a file that fails Pinnacle 21 — no labels, character variables longer than the V5 limit, columns in creation order. The corollary trap: xportr operates on attributes, so any dplyr verb applied after xportr may drop them. Do the xportr pipeline last, immediately before writing, and never `mutate()` afterwards.

Q36. What are the SAS V5 transport constraints, and how do they bite in R?

SHORT ANSWER. Dataset name and variable names at most 8 characters, uppercase, alphanumeric plus underscore; variable and dataset labels at most 40 characters; character variables at most 200 bytes; only two data types, character and 8-byte numeric. `xpt_validate()` checks these and xportr additionally asserts the output file name is 8 characters or fewer with no special characters.

FOR THE SAS PROGRAMMER. Nothing new — these are the constraints you've worked around for a decade with SUPPQUAL and split datasets. The reason to know them cold in an R context is that R imposes none of them, so nothing stops you creating a 300-character factor level or a 12-character variable name until the write step fails.

DETAIL. The practical consequences in R. **Types**: R has integers, doubles, logicals, factors, Dates, POSIXct and character. `xportr_type()` collapses all of that to character or numeric — a `Date` becomes a numeric with a `DATE9.` format attached, matching SAS's internal representation. A logical or factor left unconverted is a common failure. **Lengths**: R has no concept of variable length, so `xportr_length()` sets a `width` attribute from the spec; if your actual data exceeds the spec width the value is truncated on write, which is a silent data-loss risk — check lengths against the data, not just the spec. **200-byte limit**: long AE verbatim terms and comment fields are the usual offenders and go to SUPPQUAL. **File size**: `max_size_gb` in `xportr_write()` and `xportr_split()` exist because the FDA has practical size limits per file. **Bytes not characters**: any non-ASCII character consumes more than one byte, so a name with an accent can breach a 200-byte limit at 199 characters.

```
library(xportr)
library(dplyr)

# Check actual data widths against the spec before you write
widths <- adsl %>%
  summarise(across(where(is.character), ~ max(nchar(.x, type = "bytes"), na.rm = TRUE)))
widths

# xpt_validate() returns the list of compliance failures without writing
# xpt_validate(adsl)
```

INTERVIEW TRAP. "What happens if a character value is longer than the spec length?" It is truncated, and unless you ran with `strict_checks = TRUE` you get a warning you may not read. That's how a 210-character AE term becomes a 200-character AE term in a submitted dataset. The defensive habit is to compute maximum observed byte-lengths from the data and reconcile them against the spec as a pre-write check — a five-line piece of code that has saved real submissions.

Q37. How do you produce a define.xml, an ADRG and an SDRG in an R workflow?

SHORT ANSWER. Honestly: the define.xml usually does not come from R. Most sponsors generate it from the same spec workbook using Pinnacle 21 Enterprise or Community. The open-source R option, defineR, is explicitly experimental, supports only define.xml version 2.0.0, and its own documentation says it is not recommended for regulatory submission yet. The ADRG and SDRG are Word documents authored by a human, from a PHUSE template.

FOR THE SAS PROGRAMMER. Nothing changes here. The define generation and the reviewer's guides sit outside the programming language entirely, which is why an "R-based submission" is far less exotic than it sounds.

DETAIL. What R does contribute: metacore can read an existing define.xml into an object (`define_to_metacore()`) and exposes `xml_to_ds_spec()`, `xml_to_var_spec()`, `xml_to_value_spec()`, `xml_to_codelist()` and `xml_to_derivations()` for the pieces, which is genuinely useful for checking that the shipped define matches the shipped data. The ADRG in an R submission has extra required content that a SAS ADRG doesn't: the R version, the package versions and their source repository, how the environment is reproduced, and any known conformance gaps. In Pilot 5, for example, the inability to validate Dataset-JSON v1.1 in Pinnacle 21 was documented in the ADRG rather than hidden. The PHUSE ADRG/SDRG templates are the reference; treat the R-specific environment section as mandatory.

```
library(metacore)

# Round-trip check: does the shipped define describe the shipped data?
# mc_define <- define_to_metacore("define.xml", quiet = TRUE)
# get_keys(mc_define, "ADSL")
# get_control_term(mc_define, variable = "SEX", dataset = "ADSL")
```

INTERVIEW TRAP. If you claim "R can generate the define.xml," expect "which package, and has it been used in a filing?" The safe, credible answer names defineR, states its limitations accurately (experimental, define 2.0.0 only, authors advise against submission use), and then says what you'd actually do: generate from the spec with Pinnacle 21 and use metacore to verify the define against the data programmatically. That last part is a genuinely good idea that most teams don't do.

Q38. What did the R Consortium FDA submission pilots actually demonstrate?

SHORT ANSWER. Pilot 1 (2021–2022): tables and figures produced in R, submitted through the standard eCTD gateway. Pilot 2 (2022–2024): the same outputs delivered as a Shiny application. Pilot 3 (2023–2024): ADaM datasets as well as TLFs, all generated in R. Pilot 4 (2024–2025): alternative distribution — WebAssembly and containers. Pilot 5 (2025–2026): Dataset-JSON in place of XPT. Pilot 6 (from January 2026) and Pilot 7 are in progress.

FOR THE SAS PROGRAMMER. The important thing these prove is procedural, not technical: the FDA accepted, opened and reviewed R-generated submission packages through the normal eCTD route. Nobody doubted R could compute a mean; the question was always whether the agency could receive and reproduce it.

DETAIL, WITH DATES. **Pilot 1** — four R programs producing four TLFs on simulated CDISC pilot data, including proprietary R packages in the package; initial submission November 2021, revised submission February 2022, final FDA response March 2022. **Pilot 2** — the same outputs packaged as a Shiny app. **Pilot 3** — ADaM datasets plus TLFs generated in R. **Pilot 4** — the WebAssembly component was transmitted through the eCTD gateway on 20 September 2024 and is described as the first publicly available submission package to include a WebAssembly component; the container version went via the eCTD portal in summer 2025. Notably, FDA reviewers found the WebAssembly route easier than containers, because it needed only a browser, whereas installing container dependencies on Windows proved difficult; the viability of container technology behind the FDA firewall is still under discussion and the overall Pilot 4 report is not yet complete. **Pilot 5** — began early 2025, submitted autumn 2025, with ADaM programs writing Dataset-JSON instead of XPT and the SDTM XPTs converted directly using the `datasetjson` package; FDA requested minor rework on the use of intermediate datasets, prompting a resubmission in early January 2026, with completion hoped for spring 2026. A gap surfaced: Dataset-JSON v1.1 was not supported in Pinnacle 21, so validation was done on the XPT

versions and the gap documented in the ADRG. **Pilot 6** — kicked off January 2026 to grow the library of ADaM and display programs; explicitly not destined for FDA, laying groundwork for future pilots, and exploring AI-assisted automation. **Pilot 7** — standards, demonstrations, evaluation and teaching, with a community survey open. A separate outcome of the working group's collaboration: the FDA broadened the file formats it accepts for R packages via eCTD.

```
library(datasetjson)

# What Pilot 5 exercised: writing a dataset as Dataset-JSON rather than XPT
# ds <- dataset_json(adsl, item_id = "IG.ADSL", name = "ADSL",
#                    label = "Subject-Level Analysis Dataset")
# write_dataset_json(ds, "adsl.json")
```

INTERVIEW TRAP. People conflate the pilots with FDA policy. The pilots are demonstrations run by a volunteer working group, not a regulatory pathway, and none of them was a real marketing application. Saying "the FDA has approved R for submissions because of the pilots" is wrong on two counts. The accurate statement is that the pilots demonstrated the agency could receive, open and reproduce R-based packages, and produced public reference implementations you can copy.

Q39. So — can you submit to the FDA with R?

SHORT ANSWER. Yes, and companies do. The FDA's own Statistical Software Clarifying Statement says the agency does not require any specific software; it requires that the software produce reliable results and that the submission include the code and enough documentation for a reviewer to reproduce the analysis. What R does not change is the data format: datasets still go as SAS V5 transport files per the Study Data Technical Conformance Guide, with Dataset-JSON still at the pilot stage.

FOR THE SAS PROGRAMMER. The constraint was never the language. It was, and still is, transport format, define.xml, conformance validation and reviewer reproducibility. R clears all four, but only if you do the environment work.

DETAIL. The honest checklist for an R submission. **Datasets**: XPT v5 via xportr — check the current FDA Data Standards Catalog before assuming anything about Dataset-JSON, because that is exactly the thing that changes. **define.xml**: still typically Pinnacle 21. **Conformance**: still Pinnacle 21. **Code**: submit the R programs, and package non-CRAN or proprietary packages — `pkglite` was built for exactly this, flattening packages into a single text file that survives the eCTD. **Environment**: the ADRG must state the R version, package versions and repository snapshot; renv.lock is the artefact. **Reproducibility**: this is the real risk, and it is why Pilot 4 explored containers and WebAssembly — the reviewer has to be able to run it. **What you should not claim**: that R makes any of this easier than SAS. It makes it possible at no licence cost, with a heavier environment-management burden.

```
library(renv)
library(dplyr)

# The three lines that make the submission reproducible
# renv::snapshot()                      # writes renv.lock with exact versions
# renv::restore()                       # any reviewer or programmer rebuilds it
si <- sessionInfo()
c(R = si$R.version$version.string, platform = si$platform, collate = Sys.getlocale("LC_COLLATE"))
```

INTERVIEW TRAP. The follow-up is always "what's the hardest part?" It is not writing the derivations. It is proving that the environment that produced the results can be rebuilt years later, on someone else's machine, when CRAN has moved on. Answering "renv.lock plus an internal frozen repository snapshot plus a documented base image, and I'd test the restore before the submission, not after" shows you've thought about the part that actually fails.

Q40. Talk me through double programming: SAS production, R validation. What tolerance, and why isn't byte-identical the target?

SHORT ANSWER. The two programs must agree on the reported values within a pre-specified tolerance, using independently written code from the same spec. Byte-identical is the wrong target because the two languages legitimately differ in rounding rules, quantile definitions, sort collation and default statistical methods — chasing exact equality either forces the validator to copy the producer's choices, which destroys the independence that makes double programming worth doing, or it generates noise findings that get waived.

FOR THE SAS PROGRAMMER. You already run `PROC COMPARE` with a `CRITERION=`. The R side of this is `diffdf(base, compare, keys, tolerance)`. Same discipline, and the same rule that the validator must derive from the spec, never from the production code.

DETAIL — THE SPECIFIC PLACES SAS AND R DIVERGE, AND THEY ARE ALL REAL. **Rounding**: SAS's `ROUND()` rounds half away from zero; R's `round()` implements IEC 60559 round-half-to-even, so `round(0.5)` is 0 and `round(2.5)` is 2. Every team doing SAS-to-R validation writes a small round-half-up helper, and you should say so. **Quantiles**: SAS `PROC UNIVARIATE` defaults to `PCTLDEF=5`; R's `quantile()` defaults to `type = 7`. Medians agree; quartiles often don't. **Sorting and collation**: character sort order depends on locale in R (`LC_COLLATE`) and on the encoding/collating sequence in SAS; upper/lower case and accented characters can order differently, which changes which record `first.` / `mode = "first"` picks. Set the locale explicitly. **Missing values**: SAS sorts missing numerics as smallest; R's `NA` sorts last by default and is not less than anything — so a "minimum" over a vector with `NA` behaves differently unless you're explicit. **Survival**: `PROC LIFETEST` and `survival::survfit()` default to different confidence-interval transformations, so Kaplan-Meier CIs differ unless you match `conf.type`. **Mixed models**: degrees-of-freedom methods (Kenward-Roger, Satterthwaite) must be specified to match. **Floating point**: both use IEEE 754 doubles, so basic arithmetic agrees to about 1e-15 — that part is a non-issue and saying so shows you know where the real problems are.

```
library(diffdf)
library(dplyr)

# Round half away from zero, to match SAS ROUND()
round_sas <- function(x, digits = 0) {
  z <- abs(x) * 10^digits
  sign(x) * trunc(z + 0.5 + sqrt(.Machine$double.eps)) / 10^digits
}

stopifnot(
  round_sas(0.5) == 1,          # SAS gives 1
  round(0.5)     == 0,          # R's banker's rounding gives 0
  round_sas(2.5) == 3,
  round_sas(-0.5) == -1
)

# The comparison itself: numeric layer, keyed, with a stated tolerance
# diffdf(base = sas_results, compare = r_results,
#        keys = c("PARAMCD", "AVISIT", "TRTA", "STAT"), tolerance = 1e-8)
```

INTERVIEW TRAP. “What tolerance would you use?” There is no universal number; the answer is that tolerance is specified per output type in the validation plan before the comparison runs, and it is tighter than the display precision. If a table reports one decimal place, agreement to 1e-8 on the underlying number means the displayed value can never differ. The wrong answers are “whatever makes it pass” and “zero.” Second trap: if the validator’s independent code disagrees, the fix is not to make the validator match production — it’s to determine which is correct against the spec, which is the whole point of independence.

Module 3 — R Package Development, and Validating R in a GxP Environment

This module has two halves that a hiring manager weighs very differently. The package-development half is craft: it proves you can build something a team can depend on, and it is a genuine differentiator against candidates who only ever wrote scripts. The validation half is the one that actually gets senior people hired — a check of five currently-live senior R reqs (ClinChoice, IQVIA, Novartis x2, GSK/ViiV, verified 2026-08-02) found that **NONE OF THEM NAMED PACKAGE AUTHORSHIP, RENV, TESTTHAT, RISKMETRIC, THE R VALIDATION HUB, OR 21 CFR PART 11 BY NAME**; what they all demanded was accountability for validation governance, audit readiness and inspection readiness. So learn the tools, but answer as the person who **owns the framework**, not the person who executes the QC step.

Two honesty rules run through everything below. First: where something is community practice rather than regulation, say so out loud — "that's sponsor SOP territory, not a regulation" is one of the strongest sentences you can say in this interview. Second: where a tool is a nice-to-have rather than a hiring requirement, name it as such; a senior person who can tell the difference between a fashion and a requirement is exactly who they are trying to hire.

Section 1 — Why package at all (and when not to)

Q1. You've got a folder of R scripts that works. Why would you turn it into a package?

SHORT ANSWER. A package gives you three things a folder cannot: a namespace, so your `merge()` doesn't silently shadow someone else's; a declared dependency contract in DESCRIPTION, so the environment it needs is machine-readable; and a single command, `R CMD check`, that verifies documentation, examples and tests all still agree with the code. In a regulated setting, that last one matters most — it turns "we think it works" into a reproducible, dated, re-runnable piece of evidence.

FOR THE SAS PROGRAMMER. A folder of scripts is a directory of `%include` files. A package is a compiled, versioned, documented macro library with a formal autocall path, a header block that is checked against the actual parameters, and a test suite that runs on every change. You know the failure mode of an uncontrolled macro library: two studies quietly running different versions of `%calcbmi` because someone edited the shared copy. A package with a version number and a validated repository is the fix for exactly that.

DETAIL. Concretely, packaging buys you: namespacing (`mypkg::derive_trtdur()` is unambiguous), lazy-loading and installation semantics (users `library()` it rather than sourcing 40 files in the right order), documentation that lives next to the code and is checked for drift, a test suite with a standard runner, a version number that can be pinned, and distribution through a repository rather than a network share. It also buys you a boundary: exported functions are the public API you promise to keep stable; everything else is internal and you can refactor it freely. That boundary is what lets a validated package be maintained at all.

The cost is real: you accept `R CMD check` discipline, semantic versioning, and a release process. Do not pretend it is free.

```
# A folder of scripts
analysis/
  01_setup.R          # source() order matters, undocumented
  helpers.R           # 40 functions, no docs, no tests
  02_adsl.R

# The same thing as a package
adsl.tools/
  DESCRIPTION         # name, version, dependencies, licence
  NAMESPACE           # what is exported and what is imported
  R/                  # the functions
  man/                # generated help pages
  tests/testthat/     # the test suite
  vignettes/          # long-form worked documentation
  inst/               # anything else shipped with the install
  NEWS.md             # what changed in each version
```

INTERVIEW TRAP. The follow-up is "so should everything be a package?" No — and saying yes marks you as someone who over-engineers. Study-specific analysis code that will run once and never be reused should stay as scripts under version control; packaging it adds ceremony and a second validation surface for no benefit. The rule of thumb: **package what is reused across studies; script what is specific to one.** The moment a helper is copied into a second study, it should have become a package.

Q2. What does an R package give a clinical team that a validated SAS macro library doesn't?

SHORT ANSWER. Mostly the same things, honestly — a good SAS macro library with change control and a test corpus is a well-run system, and I'd never claim R is inherently better governed. What R adds is that the tooling for the governance is standardised and free: dependency declaration, documentation-code consistency checking, test running, coverage measurement and continuous integration are all part of the ecosystem rather than something each sponsor builds themselves.

FOR THE SAS PROGRAMMER. In SAS, your validation evidence is usually built by your organisation: a spreadsheet of test cases, a QC log, a signed memo. In R, `R CMD check`, testthat and covr produce machine-generated, timestamped artefacts of the same kind, and they are the same artefacts at every company, which makes them easier for an auditor to interpret. That standardisation is the real gain, not the language.

DETAIL. The deeper structural difference is dependency handling. A SAS macro library's dependencies are largely implicit — the SAS version and whatever else is on the autocall path. An R package declares its dependencies with version bounds in DESCRIPTION, and `renv` can capture the exact resolved set into a lockfile. That means "reproduce this analysis in five years" is a solvable problem with an explicit artefact, rather than an archaeology exercise.

The counterweight, which you should volunteer: R's dependency graphs are deeper and move faster than SAS's. A single tidyverse import can pull in dozens of transitive packages, each with its own release cadence. SAS's slower, monolithic release model is genuinely easier to keep frozen. This is a trade, not a win.

```
# DESCRIPTION — the dependency contract, machine-readable
Package: adsl.tools
Version: 1.2.0
Depends: R (>= 4.3.0)
Imports:
    dplyr (>= 1.1.0),
    rlang
Suggests:
    testthat (>= 3.0.0),
    knitr
```

INTERVIEW TRAP. "So R is better than SAS for this?" Do not take the bait — the interviewer usually has 20 years of SAS behind them. The senior answer is that the governance model is what matters and both languages can be governed well or badly; R's advantage is standardised tooling, its disadvantage is a faster-moving and deeper dependency ecosystem that demands more discipline, not less.

Section 2 — Anatomy of a package

Q3. Walk me through the files in an R package. What does each one do?

SHORT ANSWER. DESCRIPTION is the metadata and dependency contract. NAMESPACE declares what the package exports and what it imports. R/ holds the source. man/ holds generated help files. tests/ holds the test suite. vignettes/ holds long-form documentation. inst/ holds anything else you want installed alongside the package. data/ holds shipped datasets.

FOR THE SAS PROGRAMMER. DESCRIPTION is the header block for the whole library plus the setinit/ dependency statement. NAMESPACE is the autocall path plus an explicit "these macros are public, these are internal" declaration that SAS has no equivalent for. man/ is your macro header documentation, but generated from the source so it cannot drift.

DETAIL. Two of these are commonly misunderstood.

`inst/` is a staging directory: everything under `inst/` is copied to the top level of the installed package. So `inst/validation/report.Rmd` in your source tree is found at `system.file("validation", "report.Rmd", package = "adsl.tools")` after installation. This is exactly how validation packets get shipped inside the installed package so they survive deployment — `inst/validation` is the {valtools} convention.

`data/` holds `.rda` files that are lazily loaded and available by name when the package is attached. If you ship reference data used by a derivation (a lookup of visit windows, say), it belongs there and it is versioned with the package — which means a change to that lookup is a package version change, and therefore goes through change control. That is a genuinely useful property.

```
adsl.tools/
├── DESCRIPTION            # metadata, dependencies, licence, version
├── NAMESPACE              # generated by roxygen2 — do not hand-edit
├── NEWS.md                # user-facing changelog, per version
├── R/
│   ├── derive_trtdur.R
│   └── zzz.R              # .onLoad / .onAttach hooks, if any
├── man/                   # generated by roxygen2 from R/ comments
│   └── derive_trtdur.Rd
├── tests/
│   ├── testthat.R         # the runner stub
│   └── testthat/
│       ├── helper-dates.R # loaded before every test file
│       └── test-derive_trtdur.R
├── vignettes/
│   └── deriving-adsl.Rmd
├── inst/
│   └── validation/        # validation packet, installed with the package
└── data/
    └── visit_windows.rda
```

INTERVIEW TRAP. "Can I edit NAMESPACE by hand?" You can, but in any modern package you should not — it is generated by roxygen2 and carries a "do not edit by hand" header. Hand-editing it is a change-control smell, because the generated file and the source comments will disagree at the next `document()` run and the diff will look like someone tampered with the build.

Q4. Explain Imports versus Depends versus Suggests. Why is Depends usually wrong?

SHORT ANSWER. Imports means "I need this package installed and I will call into it explicitly" — it does not attach the package to the user's search path. Depends means "attach this package for the user whenever mine is attached", which pollutes their environment. Suggests means "optional — needed for tests, vignettes or an edge-case feature, but not to use the core package". Imports is the correct default for almost everything.

FOR THE SAS PROGRAMMER. Depends is the equivalent of your macro library silently adding another library to everybody's autocall path the moment they use one of your macros. It works until two libraries define the same name and the resolution order decides which one runs. Imports is the disciplined version: your code can call the other library, but the user's environment is untouched.

DETAIL. The real problem with Depends is masking. If your package Depends on dplyr, then `library(adsl.tools)` attaches dplyr, and now `filter()` in the user's session means `dplyr::filter()` rather than `stats::filter()`. You have changed the meaning of the user's code as a side effect of loading your package. In a validated analysis that is a reproducibility hazard: the result of a script now depends on the order of `library()` calls.

Use Depends only for: a minimum R version (`Depends: R (>= 4.3.0)` — this is the standard and correct use), or in the rare case where your package genuinely extends another one and users cannot work with your objects without the other package attached.

Suggests is where testthat, knitr, rmarkdown and covr go. Anything in Suggests must be guarded at runtime, because it may not be installed.

```
# In DESCRIPTION: Imports: dplyr
# In R/derive.R — call it explicitly, never rely on attachment
derive_flags <- function(dat) {
  dplyr::mutate(dat, SAFFL = ifelse(!is.na(.data$TRTSDT), "Y", "N"))
}

# A Suggests dependency must be guarded
plot_kaplan_meier <- function(dat) {
  if (!requireNamespace("survminer", quietly = TRUE)) {
    stop("Package 'survminer' is required for this plot. Please install it.",
         call. = FALSE)
  }
  survminer::ggsurvplot(...)
}
```

INTERVIEW TRAP. The follow-up is "how do you use a function from an Imports package?" Two ways: `pkg::fun()` at the call site, which is explicit and my default; or `@importFrom pkg fun` in a roxygen block, which puts an entry in NAMESPACE and lets you write `fun()` bare. Do not answer "I put `library(dplyr)` at the top of the R file" — a `library()` call inside a package's R/ directory is a genuine defect that changes global state at load time, and `R CMD check` will not always catch it.

Q5. What is NAMESPACE actually for, and how does it get generated?

SHORT ANSWER. NAMESPACE defines the package's public surface — which functions are exported and visible to users — and its import surface — which functions from other packages your code can see. It is generated by roxygen2 from `@export` and `@import*` tags in your source comments, so the file and the code cannot drift apart.

FOR THE SAS PROGRAMMER. SAS has no concept of a private macro. Everything on the autocall path is callable by anyone, so internal helpers become de facto public API and you can never safely change them. NAMESPACE gives you that boundary: exported functions are your contract, unexported ones you can rewrite freely without a breaking version bump.

DETAIL. The mechanics: you write `@export` above a function, run `devtools::document()` (which calls `roxygen2::roxygenise()`), and roxygen2 rewrites NAMESPACE and the `man/*.Rd` files. The generated file carries a header telling everyone not to hand-edit it.

Why this matters for validation: your exported API is the set of functions your validation evidence has to cover. If you export 12 functions, an auditor can ask for requirements and tests for 12 things. If you carelessly export 60 internal helpers, you have just expanded your validation scope by a factor of five for no benefit. **EXPORT DISCIPLINE IS VALIDATION SCOPE CONTROL** — that framing lands well in an interview.

```
# Generated by roxygen2: do not edit by hand

export(derive_trtdur)
export(derive_saffl)
importFrom(dplyr,mutate)
importFrom(rlang,.data)
S3method(print,adsl_summary)
```

INTERVIEW TRAP. "What happens if a user needs an internal function?" The correct answer is that they should not, and if they genuinely do, that is a signal to promote it to the public API deliberately — with documentation, tests and a version bump. Do not say "they can use the triple colon, `mypkg:::fun()`". That works, but recommending it in a validated context is wrong: `:::` reaches past the API boundary into code that carries no stability promise and may carry no validation evidence.

Q6. Should code in an R/ file do anything when the package loads?

SHORT ANSWER. Almost never. R/ files should define functions and nothing else. Code at the top level of an R/ file runs at build time, not at load time, which surprises people badly. If you genuinely need load-time behaviour, that goes in `.onLoad()` in a `zzz.R` file, and it should be minimal.

FOR THE SAS PROGRAMMER. This is the same instinct as "a macro definition file should define macros, not execute steps". A `%let` sitting loose in an autocall file is the same class of bug.

DETAIL. The classic failure: someone writes `CUTOFF_DATE <- Sys.Date()` at the top level of an R file. That value is evaluated when the package is built, frozen into the installed package, and every user thereafter gets the build date. In a validated package that is a silent data-integrity defect — the analysis silently uses a date nobody intended.

Legitimate `.onLoad()` uses are narrow: registering S3 methods for packages in Suggests, setting package options if they are not already set, or registering with an external system. Anything that reads files, hits a network, or writes to the global environment is wrong.

```
# R/zzz.R
.onLoad <- function(libname, pkgname) {
  op <- options()
  defaults <- list(adsl.tools.date_format = "%Y-%m-%d")
  toset <- !(names(defaults) %in% names(op))
  if (any(toset)) options(defaults[toset])
  invisible()
}

# WRONG — evaluated at build time, frozen into the installed package
# CUTOFF_DATE <- Sys.Date()

# RIGHT — evaluated when the user calls it
get_cutoff <- function() Sys.Date()
```

INTERVIEW TRAP. The follow-up is "what about `.onAttach()`?" It runs on `library()` and is the only place a startup message is acceptable — and even then, use `packageStartupMessage()`, not `cat()` or `message()`, so users can suppress it. A package that prints unsuppressable banner text into a log is an annoyance in interactive use and a genuine problem when the log is a regulatory record.

Section 3 — Documentation as a deliverable

Q7. How does roxygen2 work, and which tags actually matter?

SHORT ANSWER. roxygen2 reads specially formatted comments starting with `#'` above each function and generates both the `man/*.Rd` help pages and the NAMESPACE file. The tags that carry weight are `@param` for each argument, `@return` for what comes back, `@examples` for runnable examples, and `@export` to make the function public. Title and description come from the first paragraphs.

FOR THE SAS PROGRAMMER. This is your macro header block, except that `R CMD check` verifies it. If you add a parameter and forget to document it, the check fails. If your example calls the function with the wrong arguments, the check fails, because it actually runs the examples. A SAS header comment can be wrong for years and nothing will tell you.

DETAIL. The verification is the point. `R CMD check` cross-references documented arguments against the actual function signature and errors on a mismatch, and it executes every `@examples` block. That gives you a continuously-verified consistency guarantee between code and documentation — which is exactly the property an auditor is looking for when they ask "how do you know the documentation reflects the code that ran?"

For validated packages, write `@return` with real precision: not "the duration", but "an integer vector of duration in days, inclusive of both endpoints, with NA where either input is NA". That sentence is effectively a requirement statement, and it is the thing your test will assert against.

Use `\dontrun{}` sparingly and `@examplesIf` in preference — an example that is never executed is documentation that is never verified.

```
#' Derive Total Treatment Duration
#'
#' Computes treatment duration in days, inclusive of both the first and last
#' dosing dates, per the study SAP convention.
#'
#' @param start Date vector of first dose dates (ADSL `TRTSDT`).
#' @param end Date vector of last dose dates (ADSL `TRTEDT`).
#' @return An integer vector the same length as `start`, giving duration in
#'   days inclusive of both endpoints. `NA` where either input is `NA`.
#' @examples
#' derive_trtdur(as.Date("2026-01-01"), as.Date("2026-01-10"))
#' @export
derive_trtdur <- function(start, end) {
  stopifnot(inherits(start, "Date"), inherits(end, "Date"))
  as.integer(end - start) + 1L
}
```

INTERVIEW TRAP. "What's the difference between `@import` and `@importFrom`?" `@importFrom pkg fun` brings in one named function; `@import pkg` brings in everything the package exports. Prefer `@importFrom` — a blanket `@import` re-creates the masking problem that Imports was supposed to solve, just inside your package instead of the user's session. Blanket imports of two packages that share a function name is a real and hard-to-diagnose bug.

Q8. Vignettes and pkgdown — are these just nice-to-haves, or do they count as deliverables?

SHORT ANSWER. In a regulated context they can be genuine deliverables. A vignette is an executable document — R Markdown or Quarto that runs its own code when the package is built — so it is documentation that is proven to work against the version it ships with. pkgdown renders the whole documentation set, including vignettes, into a static site you can host internally as the package's controlled documentation.

FOR THE SAS PROGRAMMER. A vignette is the worked example section of your macro library user guide, except that it is re-executed on every build, so it cannot describe behaviour the code no longer has. Compare that to a Word user guide on a shared drive that was last accurate three versions ago — which is the normal state of SAS macro documentation everywhere.

DETAIL. The mechanism matters for the validation argument: `R CMD build` executes vignette code and fails the build if it errors. So a vignette that says "here is how you derive treatment duration, and here is the output" is a build-time-verified claim. That is a stronger artefact than most user documentation in any language.

Be honest about scope, though. A vignette is user documentation, not validation evidence — it demonstrates intended use, it does not demonstrate that the output is numerically correct against an independent specification. Do not present a vignette as a validation test. The two live in different places for a reason.

pkgdown is a convenience layer: `pkgdown::build_site()` turns DESCRIPTION, man/, vignettes and NEWS.md into a browsable site. It is standard practice in the R community and useful internally, but it is a documentation-hosting choice, not a compliance requirement. Say that plainly rather than implying an auditor expects it.

```
# DESCRIPTION additions for vignettes
Suggests:
    knitr,
    rmarkdown
VignetteBuilder: knitr
```

```
# vignettes/deriving-adsl.Rmd — YAML front matter
# ---
# title: "Deriving ADSL treatment variables"
# output: rmarkdown::html_vignette
# vignette: >
#   %\VignetteIndexEntry{Deriving ADSL treatment variables}
#   %\VignetteEngine{knitr::rmarkdown}
#   %\VignetteEncoding{UTF-8}
# ---
pkgdown::build_site()
```

INTERVIEW TRAP. "Does the FDA want your pkgdown site?" No. The FDA Study Data Technical Conformance Guide asks for the programs used to create ADaM datasets and produce the tables and figures supporting primary and secondary efficacy analyses, in ASCII text format, with the software identified in the ADRG. Documentation sites, vignettes and pkgdown are internal quality practice, and how you retain them is sponsor SOP territory. Claiming a regulator expects a pkgdown site is exactly the kind of overreach that costs you credibility.

Section 4 — Testing

Q9. How does testthat work, and how do you structure a test suite for derivation code?

SHORT ANSWER. Tests live in `tests/testthat/`, one file per source file by convention, each containing `test_that()` blocks with a plain-English description and one or more `expect_*()` assertions. `devtools::test()` runs them locally; `R CMD check` runs them as part of the check. Each test file runs in its own environment, so tests do not leak state into each other.

FOR THE SAS PROGRAMMER. This is your QC test case log, except executable and re-run on every change instead of once at validation time. The `test_that()` description string is the test case description; the `expect_*()` call is the acceptance criterion. The difference from a SAS validation spreadsheet is that this one re-runs automatically and fails loudly when someone breaks it two years later.

DETAIL. The structure that works for clinical derivations: one `test_that()` per behaviour, not per function. A duration function needs separate tests for the happy path, the inclusive-endpoint convention, missing dates, end-before-start, zero-length input, and type preservation. Each of those maps to a line in a requirements document.

`helper-*.R` files in `tests/testthat/` are sourced before every test file — that is where shared fixture data and helper constructors go. Files named `setup-*.R` run once before the suite and can register teardown. Use `withr::local_*()` functions to make any environment change automatically revert at the end of the test.

Set `Config/testthat/edition: 3` in DESCRIPTION. The third edition changes the comparison engine to waldo and gives much better failure messages — and it changes the semantics of `expect_equal()`, which matters for the next question.

```
# tests/testthat/helper-dates.R
d <- function(x) as.Date(x)

# tests/testthat/test-derive_trtdur.R
test_that("duration is inclusive of both endpoints per the SAP", {
  expect_equal(derive_trtdur(d("2026-01-01"), d("2026-01-10")), 10L)
})

test_that("same-day start and stop is one day, not zero", {
  expect_equal(derive_trtdur(d("2026-01-01"), d("2026-01-01")), 1L)
})

test_that("missing dates propagate rather than erroring", {
  expect_true(is.na(derive_trtdur(d(NA), d("2026-01-10"))))
})

test_that("non-Date input is rejected", {
  expect_error(derive_trtdur("2026-01-01", d("2026-01-10")))
})
```

INTERVIEW TRAP. "How many tests is enough?" The wrong answer is a number. The right answer is that the test suite should cover every documented behaviour and every requirement, plus every bug ever found — a fixed bug without a regression test will come back. The framing that sounds senior: **tests are traceable to requirements, not to lines of code.** That is the bridge into {valtools} and into the coverage question below.

Q10. Two derivations produce numbers that differ in the fifteenth decimal place. Is that a discrepancy?

SHORT ANSWER. No, and treating it as one is a serious error. Doubles are binary floating point, so decimal values like 0.1 have no exact representation and arithmetic accumulates tiny representation error. The correct comparison is a tolerance-based one — `all.equal()` in base R, or `expect_equal()` in testthat — not `identical()`. Byte-identical output is the wrong target.

FOR THE SAS PROGRAMMER. You have met this exactly: `PROC COMPARE` reports a difference, you look at it, and it's 1e-16 on a mean. SAS's `FUZZ=` option and the `MAXDIF` column exist for the same reason. Same physics, different tooling — R just makes you choose the tolerance explicitly rather than defaulting it.

DETAIL. The precise mechanics: `identical()` is an exact structural comparison — same type, same attributes, same bits — and it will return FALSE for `0.1 + 0.2` versus `0.3`. `all.equal()` compares with a tolerance, by default `sqrt(.Machine$double.eps)`, roughly 1.5e-8, and it is relative for larger values and switches toward absolute as values approach zero. Note `all.equal()` returns TRUE or a character description of the differences, so you almost always want `isTRUE(all.equal(x, y))` in a conditional — using it bare in an `if()` is a classic bug.

In testthat's third edition, `expect_equal()` and `expect_identical()` both run on waldo and differ only in that `expect_equal()` applies `testthat_tolerance()`. A consequence worth knowing: under 3e, `expect_equal(1, 1L)` passes, because tolerance-based comparison ignores the integer-versus-double distinction. If type matters — and for an ADaM variable it often does — use `expect_identical()`.

The governance point, which is the one to lead with in an interview: **THE TOLERANCE IS A DOCUMENTED DECISION, NOT A DEFAULT YOU INHERIT.** Your validation plan should state what agreement threshold constitutes a match for each class of output, and why. "Numerically equal within 1e-8 relative" is a defensible specification. "The files are byte-identical" is not — it will fail on a different machine, a different BLAS, or a different R build, for reasons that have nothing to do with correctness.

```
x <- 0.1 + 0.2

identical(x, 0.3)                 # FALSE
isTRUE(all.equal(x, 0.3))         # TRUE
print(x - 0.3, digits = 20)       # 5.5511151231257827e-17

# testthat 3rd edition
expect_equal(x, 0.3)                    # passes, tolerance = testthat_tolerance()
expect_identical(x, 0.3)                # fails
expect_equal(x, 0.3, tolerance = NULL)  # fails — exact comparison

# An explicit, documented tolerance for a derived endpoint
expect_equal(auc_observed, auc_expected, tolerance = 1e-6)
```

INTERVIEW TRAP. "So how do you compare a whole PD versus QC dataset?" Do not say "PROC COMPARE has no R equivalent" — that is wrong. `all.equal()` on data frames, `waldo::compare()` for a readable structural diff, `diffdf::diffdf()` for a PROC-COMPARE-shaped report with by-variable keys, and `arsenal::comparedf()` are all used in practice. The deeper trap is the second follow-up: **what do you do when the difference is 1e-9 but it flips a rounding boundary and changes a displayed value?** Then it is a real discrepancy despite being numerically tiny, and the answer is that rounding for display must be specified and tested separately from the underlying derivation. That answer marks you as someone who has actually done this.

Q11. What is a snapshot test and when would you use one in clinical work?

SHORT ANSWER. A snapshot test records the output of an expression to a file on first run, then on every subsequent run compares against that recorded file and fails if it changed. It is for output that is tedious to assert on field by field — printed table layouts, error messages, complex structures. You review and deliberately accept changes with `testthat::snapshot_accept()`.

FOR THE SAS PROGRAMMER. It is the same idea as keeping a reference RTF or listing and comparing new output against it, except the reference file is version-controlled next to the test, and the diff is shown to you in the test failure rather than found by eye.

DETAIL. Snapshots live in `tests/testthat/_snaps/` as `.md` files and are committed to the repository. When output changes, the test fails and shows a diff; you inspect it, and if the change is intended, you accept it — which updates the file, and that update appears in the pull request diff for a reviewer to approve. That review step is the valuable part: **an intentional output change becomes a reviewed, attributable commit.** That is a real audit-trail property, and it is worth saying that way.

They are excellent for TLF shell rendering and for error message wording. They are poor for anything with a floating-point number in it — snapshots are string comparisons, so a 1e-16 difference produces a spurious failure. Test numbers with tolerance-based expectations; snapshot the layout.

```
test_that("the demographics table header matches the shell", {
  expect_snapshot(print(build_demog_table(adsl_test)))
})

test_that("input validation gives a helpful message", {
  expect_snapshot(derive_trtdur("not a date", Sys.Date()), error = TRUE)
})

# When output legitimately changes:
#   testthat::snapshot_review()   # inspect the diff interactively
#   testthat::snapshot_accept()   # accept, then commit the updated _snaps file
```

INTERVIEW TRAP. "Isn't that just recording whatever the code does today, including its bugs?" Yes — that is the honest weakness, and admitting it is the good answer. A snapshot proves stability, not correctness. Its first value comes from a human reviewing the initial snapshot against a specification. So: snapshot tests are a change-detection control, not a correctness control, and your validation evidence needs both.

Q12. You have 100% code coverage. Is the package validated?

SHORT ANSWER. No. Coverage measures which lines of code were executed during the test run. It says nothing about whether the assertions were meaningful, whether the specification was right, or whether the untested input combinations behave correctly. You can reach 100% coverage with tests that assert nothing at all.

FOR THE SAS PROGRAMMER. It is the difference between "every macro was called during QC" and "every derivation was checked against the spec". You already know that the first one is easy and nearly worthless. Coverage is the executed-lines number, not the correctness number.

DETAIL. `covr::package_coverage()` instruments the package, runs the test suite, and reports the proportion of expressions executed, per file and overall. `covr::report()` gives a browsable HTML view with uncovered lines highlighted. It is genuinely useful — a low coverage number is a reliable

danger signal, and looking at which lines are red usually finds a whole branch nobody tested. But high coverage is not evidence of correctness.

Two specific gaps to name if pressed. First, line coverage does not equal branch or path coverage — a single line with three logical conditions counts as covered when one of them is exercised. Second, coverage is blind to the input space: you can execute every line of a date derivation without ever passing it a partial date, a date at a daylight-saving boundary, or a leap day.

What actually constitutes evidence is traceability: each documented requirement maps to at least one test, and each test's assertions are reviewed against the specification by someone who did not write the code. That is the {valtools} model, and it is the thing to say when someone waves a coverage percentage at you.

```
covr::package_coverage()
# adsl.tools Coverage: 96.42%
# R/derive_trtdur.R: 100.00%
# R/build_table.R:    88.10%

covr::report()  # HTML view, uncovered lines in red

# 100% coverage, zero validation value:
test_that("it runs", {
  expect_no_error(derive_trtdur(as.Date("2026-01-01"), as.Date("2026-01-10")))
})
```

INTERVIEW TRAP. "What coverage target do you set?" Any number you give will be treated as arbitrary, because it is. The senior answer inverts the question: you set a floor to catch neglect — many teams use 80 or 90 percent as a CI gate — but you never treat hitting it as evidence of validation, and you require requirement-to-test traceability separately. If the interviewer's SOP mandates a number, comply with it and say so; just do not confuse the metric with the outcome.

Section 5 — Checking, versioning, release

Q13. What does R CMD check do, and what do NOTE, WARNING and ERROR actually mean?

SHORT ANSWER. `R CMD check` runs a large battery of static and dynamic checks on a built package: it installs it, verifies documentation matches the code, runs every example, runs the test suite, checks the dependency declarations, and inspects for common defects. Findings come in three severities — ERROR means something broke outright, WARNING means a genuine problem you must fix, NOTE means an advisory that may or may not need action.

FOR THE SAS PROGRAMMER. There is no SAS equivalent, and that is worth saying. The closest analogue is a log scan for ERROR, WARNING and suspicious NOTE messages — which you have certainly built or run — but `R CMD check` goes further, because it also verifies that the documentation describes the parameters the code actually has, and it executes the examples and the test suite as part of the same pass.

DETAIL. Run it as `R CMD check --as-cran` on the built tarball, or `devtools::check()` from within R. The `--as-cran` flag turns on the stricter checks CRAN applies to incoming submissions. `rcmdcheck::rcmdcheck()` returns the results as a structured object with `errors`, `warnings` and `notes` vectors, which is what you use in CI to fail the build at a chosen threshold.

Typical findings and what they mean in practice: an ERROR from a failing example or test is code that does not work. A WARNING for an undocumented argument means your documentation and signature have drifted. A NOTE for "no visible binding for global variable" usually means you used a bare column name in a dplyr pipeline without `.data$` — cosmetic in appearance, but it genuinely indicates the checker cannot see where that symbol comes from. A NOTE about non-standard files at top level means build hygiene.

```
R CMD build adsl.tools
R CMD check --as-cran adsl.tools_1.2.0.tar.gz
```

```
devtools::check()

# In CI, fail the build on anything above a chosen severity
rcmdcheck::rcmdcheck(error_on = "warning")
```

INTERVIEW TRAP. "Is a NOTE acceptable in a validated package?" The nuanced answer wins here. There is no regulation that mentions `R CMD check` at all, so "acceptable" is defined by your own SOP. The defensible position: zero ERRORs and zero WARNINGs, and every remaining NOTE individually reviewed and documented with a justification in the validation record. An unexplained NOTE is not a compliance failure; an unexamined one is a quality-system failure, because it means nobody looked.

Q14. Why does R CMD check need to be clean for a package you intend to validate?

SHORT ANSWER. Because it is the cheapest objective evidence you have that the package is internally consistent — code, documentation, examples and tests all agree — and it is re-runnable on demand by anyone, including an auditor. A clean check on a specific version, captured with a timestamp and an environment record, is a validation artefact.

FOR THE SAS PROGRAMMER. Think of it as the difference between a QC log you generated once at validation time and one you can regenerate in front of an inspector. The second is far stronger. `R CMD check` output is exactly that: reproducible, machine-generated, and it names the R version and platform it ran on.

DETAIL. Be precise about what the clean check does and does not prove. It proves the package builds, installs, its documentation matches its code, its examples run, and its own test suite passes. It does not prove the derivations are scientifically or clinically correct — that is what your requirements, test cases and independent review provide. Conflating the two is the single most common overreach in this area, and an auditor will find it.

Where it belongs in the evidence package: the check log is your operational qualification evidence for the package itself — objective evidence that the installed software functions as intended in the target environment. Store it with the R version, the platform, the package version, the date, and the identity of whoever ran it. In practice you get this for free by running the check in CI on a tagged release, which also gives you the attribution.

```
* using R version 4.4.1 (2026-06-14)
* using platform: x86_64-pc-linux-gnu
* checking for file 'adsl.tools/DESCRIPTION' ... OK
* checking package dependencies ... OK
* checking Rd \usage sections ... OK
* checking examples ... OK
* checking tests ... OK
  Running 'testthat.R'
Status: OK
```

INTERVIEW TRAP. "Which regulation requires a clean R CMD check?" None. If you say or imply otherwise you have handed the interviewer a reason to distrust everything else you said. The correct answer: no regulation names R CMD check. ICH E9 says software used for data management and statistical analysis should be reliable and that documentation of appropriate testing procedures should be available; a clean check is one way a sponsor chooses to satisfy its own SOP for that. Requiring it is sponsor policy, and I'd write that policy — but I'd never call it a regulatory requirement.

Q15. How do you version a package, and what goes in NEWS.md?

SHORT ANSWER. Semantic versioning: MAJOR.MINOR.PATCH, where MAJOR increments on a breaking change to the public API, MINOR on backward-compatible new functionality, and PATCH on backward-compatible bug fixes. NEWS.md records what changed in each released version, written for the user, grouped by version heading.

FOR THE SAS PROGRAMMER. This is the version number on your validated macro library, but with a rule attached that tells a downstream user how scared to be. A PATCH bump means "install it, nothing you wrote will break". A MAJOR bump means "read the notes, your calling code may need changes". SAS macro libraries rarely encode that promise in the number, so every update triggers the same full re-check. Semantic versioning lets you scale the impact assessment to the actual risk.

DETAIL. The connection to change control is the point. Your SOP can key impact assessment off the version component that changed: a PATCH may need a targeted regression test on the affected function, a MINOR needs the new functionality validated but existing evidence stands, a MAJOR needs a re-assessment of every downstream study using it. That is a proportionate, risk-based change-

control policy expressed in three digits — and "risk-proportionate" is the language ICH E6(R3) uses about computerised systems generally.

Development versions conventionally carry a fourth component, `1.2.0.9000`, to make it obvious a build is not a release. Do not let a `.9000` version anywhere near a study deliverable.

NEWS.md should say what changed and why it matters to the caller, and should explicitly flag anything that alters numerical output. A bug fix that changes a derived value is, from a study's point of view, a potentially reportable event — the entry must make that unmissable.

```
# adsl.tools 1.2.0

## Breaking changes
* None.

## New features
* `derive_saffl()` gains a `dose_var` argument (#41).

## Bug fixes
* `derive_trtdur()` no longer returns 0 when start and stop dates are equal;
  it now correctly returns 1, per the SAP inclusive-endpoint convention.
  **This changes derived TRTDURD values.** Studies using versions 1.0.0 to
  1.1.2 must assess impact (#38).

# adsl.tools 1.1.2
* Documentation only; no change to derived values.
```

INTERVIEW TRAP. "You fixed a bug that changes numbers, and three ongoing studies use that package. What now?" Do not answer "I bump the version and tell them". The expected answer is a change-control answer: raise it through the quality system, assess impact per study with the statistician, decide per study whether to adopt the new version or stay pinned to the old one with a documented rationale, and — if a study has already reported results computed with the defective version — that becomes a deviation and potentially a data-integrity issue, not a software issue. Notice that the correct answer routes through people and process, not through git.

Section 6 — Repositories, environments and reproducibility

Q16. Where do packages come from in a controlled environment? Do you install from CRAN?

SHORT ANSWER. Not directly, in a well-run setup. Users install from an internal repository that the organisation controls — a curated mirror or subset of CRAN, plus the organisation's own packages. That gives you a single point where risk assessment, approval and version freezing happen, and it means what a programmer installs today is what the SOP approved, not whatever CRAN published this morning.

FOR THE SAS PROGRAMMER. This is the difference between everyone having write access to the shared macro library and having a controlled, released library that a change-control process publishes into. You already know which of those survives an audit.

DETAIL. The usual mechanics. Posit Package Manager is the common commercial tool: it mirrors CRAN, Bioconductor and internal sources, serves pre-built binaries, and — critically — supports snapshots, so you can point R at a repository URL frozen to a specific date and get exactly the package versions that existed on that date. The URL form is a dated path under the CRAN repo, and the frozen URL is recorded in the renv lockfile, so the pinning is captured in the project.

The open alternatives are a plain internal CRAN-style repository built with `drat` or `miniCRAN`, or simply hosting the approved tarballs and setting `options(repos = ...)`. These work; they are just more manual to curate.

Whichever you use, the governance layer is the same and it is the layer they are hiring for: a documented process for how a package gets into the approved repository — who requests it, what risk assessment is run, who approves, and how the decision and its evidence are retained. The tool is interchangeable; the process is the deliverable.

```
# Point the session at a date-frozen snapshot
options(repos = c(CRAN = "https://packagemanager.posit.co/cran/2026-01-15"))
install.packages("dplyr")   # gets the version that existed on 2026-01-15

# Or an internal repository of approved packages only
options(repos = c(INTERNAL = "https://rpkgs.sponsor.internal/approved"))
```

INTERVIEW TRAP. "Doesn't a frozen repository mean you never get security fixes?" Yes, and that is the real trade — you should raise it before they do. The answer is a periodic, scheduled re-baselining: the snapshot date is advanced deliberately on a defined cycle, the delta is impact-assessed, and studies in flight stay on their pinned lockfile while new studies start on the new baseline. Freezing forever is not a strategy; freezing with a scheduled review is.

Q17. Explain renv. How does it actually work?

SHORT ANSWER. renv gives each project its own private package library instead of a shared system library, and records the exact packages and versions in a JSON lockfile called `renv.lock`. `renv::snapshot()` writes the lockfile from what the project currently uses; `renv::restore()` reinstalls

exactly what the lockfile describes. A global cache means the same package version is only downloaded and compiled once across all projects.

FOR THE SAS PROGRAMMER. The closest analogue is pinning a study to a specific SAS version plus a specific frozen macro library, and writing both into the study documentation. renv makes that machine-readable and machine-restorable rather than a sentence in a plan that someone has to honour manually.

DETAIL. The mechanism: `renv::init()` creates three things — a project library at `renv/library`, the lockfile, and a project-level `.Rprofile` that activates renv whenever R starts in that directory. That `.Rprofile` is what makes it stick; without it renv is inert.

The lockfile has two top-level sections. `R` records the R version and the repository URLs — this is where a Posit Package Manager frozen URL gets captured. `Packages` records one entry per package with its name, version, source, repository and a hash, which is enough to reinstall it from CRAN, Bioconductor, or a git remote. For GitHub sources it additionally records the remote host, username, repo, ref and the resolved commit SHA, which is what makes an internal package installed from git genuinely pinned.

What to commit: `renv.lock`, `.Rprofile`, `renv/settings.json` and `renv/activate.R`. Not the library itself.

```
{
  "R": {
    "Version": "4.4.1",
    "Repositories": [
      { "Name": "CRAN", "URL": "https://packagemanager.posit.co/cran/2026-01-15" }
    ]
  },
  "Packages": {
    "dplyr": {
      "Package": "dplyr",
      "Version": "1.1.4",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "<md5-style fingerprint of the package record>"
    },
    "adsl.tools": {
      "Package": "adsl.tools",
      "Version": "1.2.0",
      "Source": "GitHub",
      "RemoteHost": "git.sponsor.internal",
      "RemoteUsername": "biostat",
      "RemoteRepo": "adsl.tools",
      "RemoteRef": "v1.2.0",
      "RemoteSha": "<full commit SHA>"
    }
  }
}
```

INTERVIEW TRAP. “Is renv a regulatory requirement?” No. Nothing requires renv, and no regulator names it. It is a widely-adopted community tool that helps you satisfy a requirement you do have — that the software used is documented including version, per the FDA’s 2015 Statistical Software Clarifying Statement, and that analyses are traceable and reproducible. Say it that way. Also worth knowing: none of the five senior R reqs surveyed for this pack named renv explicitly, so treat it as craft that supports the governance answer, not as the answer itself.

Q18. It's 2031 and a regulator asks you to reproduce a 2026 analysis. Does renv get you there?

SHORT ANSWER. Partly. renv restores your R packages, and that is the biggest part of the problem. It does not restore the R version itself, the operating system, system libraries, compilers, or external tools like Pandoc. For a genuine long-horizon reproduction you need renv plus a pinned environment — in practice, a container image.

FOR THE SAS PROGRAMMER. You already know that "we still have the code" is not the same as "we can still run it". A SAS 9.4 program needs a SAS 9.4 installation on an OS it supports and licence keys that have not expired. R has the same problem with a different shape: the code is fine, the packages are recoverable, and the thing that bites you is the platform underneath.

DETAIL. renv's own documentation is honest about the boundaries, and quoting them is a strong move. The R version is recorded in the lockfile but not managed — renv runs inside R, so it cannot switch R versions for you; tools like rig do that. Pandoc is not bundled, so restoring rmarkdown does not guarantee identical document rendering. Operating system, system library versions and compiler versions are explicitly out of scope, and the documentation recommends Docker as the complement. And there is a practical failure mode: if a package was originally installed as a binary and that binary is no longer served, renv falls back to installing from source, which can fail on missing system prerequisites.

So the layered answer, which is the one that sounds like someone who has thought about it: the code is in version control at a tagged commit; the packages are pinned in the lockfile against a date-frozen repository; the platform is pinned in a container image stored in a registry; and the whole thing is documented in the study's analysis environment record. Then reproduction is pulling one image and running one command.

Add the honest caveat: even that is not perfect over a decade. Container runtimes change, registries get migrated, and hardware architecture shifts — the move to arm64 broke a lot of assumptions. Long-term reproducibility is a retention and periodic-verification problem, not just a tooling problem. If you say that, you sound like someone who has been through an inspection.

```
# rocker/r-ver pins the R version and defaults to a date-matched repository
FROM rocker/r-ver:4.4.1

RUN apt-get update && apt-get install -y --no-install-recommends \
    libcurl4-openssl-dev libssl-dev libxml2-dev pandoc \
 && rm -rf /var/lib/apt/lists/*

WORKDIR /study
COPY renv.lock renv.lock
RUN R -e "install.packages('renv'); renv::restore()"
COPY . .
CMD ["Rscript", "run_analysis.R"]
```

INTERVIEW TRAP. "Do you archive the packages themselves?" Many sponsors do, and it is the belt-and-braces answer: retain the source tarballs of every package used, alongside the lockfile, in the study archive. Relying on a public repository still existing in ten years is a risk you do not control. Whether your organisation requires that is sponsor SOP territory — but volunteering it shows you think about the failure mode rather than the happy path.

Q19. Describe the git workflow you'd put in place for a clinical R codebase.

SHORT ANSWER. Protected main branch that nobody commits to directly; short-lived feature branches; every change enters main through a pull request that requires at least one approving review and a passing CI run; releases are tagged with the version number. The value in a regulated setting is that this produces an attributable, timestamped, immutable record of who changed what, why, and who approved it.

FOR THE SAS PROGRAMMER. This replaces the folder-with-dates convention and the emailed "please review my program" — and it replaces them with something an auditor can read. Every merged change carries an author, a reviewer, a timestamp, a linked issue explaining why, and the exact diff. That is a stronger audit trail than most SAS environments have for programs, and it is worth saying so.

DETAIL. The mapping to quality-system concepts is what makes this an interview answer rather than a tooling answer. The pull request is the change record: the description states the reason for change, the linked issue traces it to a request or a defect, the diff is the change itself, the CI run is the objective evidence that checks and tests passed, and the approval is the reviewer's signature. Branch protection is the control that makes the process non-optional rather than a convention people follow when they remember.

Be careful with one claim. A git history is tamper-evident, not tamper-proof: history can be rewritten with a force push, which is precisely why you disable force-push on protected branches and rely on the server-side record. If asked whether git satisfies 21 CFR Part 11 electronic signature requirements, the answer is that a git commit is not an electronic signature in the Part 11 sense — that is a separate question, covered later in this module.

```
main            protected; no direct pushes; linear history
 └─ feature/38-trtdur-inclusive-endpoints
      commit  "fix: TRTDUR inclusive of both endpoints (#38)"
      PR      links issue #38, states reason for change
      CI      R-CMD-check + tests + coverage — all green
      review  approved by a second programmer
      merge   squashed into main, attributable and timestamped
      tag     v1.2.0
```

INTERVIEW TRAP. "Who reviews the reviewer?" Or its harder cousin: "what stops someone approving their own pull request?" The answer is a repository setting — require approval from someone other than the author, and require a minimum number of approvals — plus an SOP that names who is qualified to review. If you answer only "we agreed as a team", you have described a convention, not a control. Auditors are trained to find exactly that gap.

Q20. What runs in CI, and why does it matter for validation?

SHORT ANSWER. At minimum: `R CMD check` including the test suite, on every push and every pull request, on the R version and platform the organisation actually uses. Usually also code coverage and a lint or style check. It matters because CI produces machine-generated, timestamped, attributable evidence that the checks were run — as opposed to someone asserting they ran them locally.

FOR THE SAS PROGRAMMER. Think of it as an automated, unavoidable QC run that fires on every change, with logs retained. The step change from "the programmer ran the checks" to "the system ran the checks and the record exists" is exactly the step change quality systems are built around.

DETAIL. In practice this is GitHub Actions or GitLab CI. The R community has maintained standard actions — `r-lib/actions` — that install R, resolve dependencies from DESCRIPTION, and run the check, so this is a short configuration file rather than a project.

Two governance points worth making. First, run the check on the specific R version your validated environment uses, not just the latest release — a green check on R-devel tells you nothing about your production environment. Second, retain the run logs: they are the objective evidence, and the retention period is a records-management decision for your SOP, not something the CI tool decides for you.

And name the limit honestly: CI proves the package's own tests passed. It does not prove the outputs are clinically correct, and it is not a substitute for independent verification of study results.

```
name: R-CMD-check
on: [push, pull_request]

jobs:
  R-CMD-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: r-lib/actions/setup-r@v2
        with:
          r-version: '4.4.1'          # the validated environment's version
      - uses: r-lib/actions/setup-r-dependencies@v2
        with:
          extra-packages: any::rcmdcheck, any::covr
      - uses: r-lib/actions/check-r-package@v2
      - name: Coverage
        run: Rscript -e 'covr::codecov()'
```

INTERVIEW TRAP. "Is CI required?" No — no regulation mentions continuous integration. But here is the framing that lands: a regulator does not care how the evidence was produced, only that it exists, is attributable, and is contemporaneous. CI happens to produce evidence with all three of those properties automatically, which is why it is good practice rather than why it is mandated. If you describe CI as a compliance requirement you will be corrected.

Section 7 — Validation and GxP: the part that gets you hired

Q21. Is R validated?

SHORT ANSWER. That question can't be answered as asked, and the best thing I can do is unpack it — because the ambiguity is where most confusion in this space lives. Software isn't validated in the abstract; it is validated for an intended use within a defined process. So the question splits into three: is the **software** fit for purpose and developed under an adequate lifecycle, is the **environment** it runs in installed and operating as specified, and is the **output** of a specific analysis verified as correct. Those are three different activities with three different owners and three different evidence sets.

FOR THE SAS PROGRAMMER. You have lived this. Nobody validated SAS — SAS Institute did their development lifecycle work, your organisation qualified the installation, and your team verified each study's outputs through double programming and review. "Is SAS validated?" was never the question either; the industry just stopped asking it because SAS is familiar. R gets the question because it is unfamiliar, not because it is different.

DETAIL. Spelling out the three layers, because this is the answer that gets you hired:

The software. For base R and the recommended packages, the R Foundation publishes a guidance document — "R: Regulatory Compliance and Validation Issues", current version dated 12 March 2026 — describing R Core's software development lifecycle, source-code management, and the regression and validation test suite that ships in the `tests` subdirectory of the source tarball. Crucially, that document is explicit that it applies **solely** to base R plus recommended packages, and explicitly not to CRAN or any other add-on package. It is also not an FDA-endorsed document; it says of itself that it is not prescriptive and confers no legal obligation, and it states that a significant obligation rests on the end-user organisation to define and enforce its own SOPs. For add-on packages you need your own risk-based assessment, which is where the R Validation Hub work comes in.

The environment. Installing R, the approved packages and the platform, then producing objective evidence that the installation matches specification and functions as intended. This is installation and operational qualification, and it is your organisation's job.

The output. Verifying that a specific study's derived datasets and TLFs are correct — independent programming, code review, output QC. Also your organisation's job, per your SOPs.

```
Layer            Owner                        Evidence
---------------  ---------------------------  -------------------------------------
Base R software  R Core / R Foundation        Published SDLC guidance; the `tests/`
                                              regression suite in the source tarball
Add-on packages  Sponsor (risk assessment)    Package risk assessment + evidence held
                                              per SOP; supplemented by own testing
Environment      Sponsor IT / Biostat         IQ/OQ records; installed inventory;
                                              `make check` output; access controls
Output           Study team                   Independent programming, review, QC
                                              records per SOP
```

INTERVIEW TRAP. The interviewer may push: "so you're saying R isn't validated?" Do not get defensive and do not oversell. The line to have ready: "R is no more and no less validatable than SAS. The FDA's own 2015 Statistical Software Clarifying Statement says the agency does not require any specific software for statistical analyses, and that statistical software is not explicitly discussed in Title 21 —

what it asks is that the software used be fully documented in the submission, including version and build. The obligation to establish fitness for purpose sits with the sponsor either way."

Q22. What is the risk-based approach to R packages, and what is the R Validation Hub?

SHORT ANSWER. The R Validation Hub is a cross-industry collaboration, working under the R Consortium, that produced a white paper called "A Risk-based Approach for Assessing R package Accuracy within a Validated Infrastructure". Its core proposition is that you cannot validate thousands of CRAN packages exhaustively, so instead you assess each package's risk using observable quality indicators, and scale your own testing effort in inverse proportion to the confidence those indicators give you.

FOR THE SAS PROGRAMMER. The nearest analogue is vendor qualification. You did not test every PROC in SAS; you relied on the vendor's development lifecycle, and you spent your own testing effort on your macros and your outputs. The Hub's framework is the same logic applied to a supplier who is a distributed open-source community rather than a company — you assess the evidence of their process quality instead of auditing a vendor.

DETAIL. The framework has two halves. First, assess the package: how actively is it maintained, does it have tests, does it have documentation and vignettes, how quickly are bugs closed, how many reverse dependencies rely on it, does it have a source repository and a maintainer. Second, assess your intended use: a package used to render a plot in an exploratory review carries different risk from one computing the primary efficacy estimand. Risk is the product of the two, and the depth of your own qualification testing follows from it.

This is the same logic ICH E6(R3) applies to computerised systems generally — validation approaches based on risk assessment, with scope proportionate to system criticality. Important nuance: risk-based does not mean low-risk systems skip validation; it means the depth matches the consequence of failure. Say that sentence and you will sound like you have read the guideline, because you have.

Two things to be honest about. The Hub's own materials note that companies diverge on what constitutes the correct approach, and that these tools are designed to support decision-making, not to replace inspection by the reviewing party. And the framework is community consensus, not regulation — no regulator has adopted it.

```
Risk score of the package        LOW ──────────────────────────► HIGH
(maintenance, tests, docs,
 bug response, community)

Criticality of your use          LOW ──────────────────────────► HIGH
(exploratory plot ... primary efficacy derivation)

Your qualification effort:
  low risk  + low criticality  → accept on assessment, record the decision
  low risk  + high criticality → assessment + targeted testing of the functions used
  high risk + high criticality → full independent qualification, or don't use it
```

INTERVIEW TRAP. "Does the FDA accept the R Validation Hub framework?" There is no FDA acceptance of any framework, and claiming otherwise is a serious error. What is true and useful: the R Consortium's Submissions Working Group has completed several pilot submissions through the FDA's eCTD gateway — Pilot 1 delivered tables and figures produced in R, Pilot 2 delivered them inside a Shiny application, Pilot 3 added ADaM datasets generated in R, Pilot 4 covered WebAssembly and containers, and Pilot 5 used Dataset-JSON in place of XPT. Those pilots demonstrate the agency can

receive and review R-based submissions. That is a very different claim from "the FDA endorses a validation framework", and the difference is one an experienced interviewer will test.

Q23. What does riskmetric actually score? And is it still the current tool?

SHORT ANSWER. riskmetric is the R Validation Hub's package that evaluates an R package against observable quality criteria and converts them into scores. The workflow is three steps: reference the package, assess it, score it. Worth knowing as of now — riskmetric has moved to **maintenance-only** status and the Hub has shifted development to a successor called val.meter, described as the next generation of riskmetric.

FOR THE SAS PROGRAMMER. It is an automated supplier-assessment scorecard. It reads the things you would look at manually when deciding whether to trust a third-party component — is it maintained, is it tested, is it documented, do bugs get fixed — and turns them into a record you can file.

DETAIL. The assessments riskmetric computes are concrete and worth being able to name a few of: code coverage, number and depth of dependencies, downloads over the last year, whether exported functions have help pages, whether the export namespace is documented, presence of a bug reports URL, presence of examples, presence of a named maintainer, presence of a NEWS file and whether it is current, presence of a source control link, presence of vignettes, presence of a website, the status of the last 30 bug reports, the licence, R CMD check results, remote check results, reverse dependencies, and codebase size.

Note carefully what that list is: process and community-health indicators. **Not one of them assesses whether the statistical method the package implements is correct.** riskmetric cannot tell you whether a survival function is right. It tells you whether the project that produced it looks well run. Being able to state that limitation cleanly is the single most valuable thing about knowing this tool.

The Hub also ships riskassessment, a Shiny front end that stores results in a database with organisation-level metric weighting and endorse/prohibit rules, and riskscore; and it runs a Regulatory Repository initiative described as a public resource for a regulatory-ready R package ecosystem with standardised, transparently assessed, publicly available quality measures. The Hub is explicit that the weighting into a single overall score is not settled — devising the best way to summarise risk into one number is listed among its open community questions.

```
library(riskmetric)

pkg_ref("dplyr") |>
  pkg_assess() |>
  pkg_score()

# Assess a set of packages and store the result as a dated record
pkg_ref(c("dplyr", "survival", "admiral")) |>
  pkg_assess() |>
  pkg_score() |>
  write.csv("evidence/package_risk_2026-08-02.csv", row.names = FALSE)
```

INTERVIEW TRAP. "So a high riskmetric score means the package is validated?" No, on two counts. First, it measures process health, not numerical correctness. Second, a score is an input to a decision, not the decision — someone qualified has to review it, weigh it against the criticality of the intended use, and record an approval. If you present a score as a conclusion you have automated your way past the judgement that a quality system exists to capture. The senior version of this answer: riskmetric produces evidence for a decision that a human owns.

Q24. What is valtools, and how does validation-as-a-package work?

SHORT ANSWER. valtools is a validation framework for R packages used in clinical research, produced by a PHUSE working group. Its central idea is that you declare formal requirements, write test cases against them, write test code that executes those cases, and explicitly map each requirement to its tests — then generate a signed validation report from that structure. The validation material lives inside the package, so it travels with the code.

FOR THE SAS PROGRAMMER. This is your validation plan, requirements specification, test scripts, traceability matrix and validation summary report — except they live in the repository next to the code, are executed rather than described, and the traceability matrix is generated from the actual artefacts rather than maintained by hand in Excel. You know how quickly a hand-maintained traceability matrix goes stale. That is the problem this solves.

DETAIL. The workflow: `vt_create_package()` creates a new package with validation infrastructure already in place (it wraps `usethis::create_package()` and adds a validation folder under vignettes); `vt_use_validation()` retrofits an existing package. A `validation.yml` config file holds shared settings including the names for the signature page. Then `vt_use_req()`, `vt_use_test_case()` and `vt_use_test_code()` add requirements, test cases and test code; `vt_use_change_log()` creates a change log; `vt_use_report()` creates the report R Markdown that renders into the final validation report. Helper functions `vt_path()` and `vt_file()` resolve paths relative to the validation directory, and the `vt_scrape_*` family pulls the structure back out as data frames for building the traceability tables.

For post-installation availability, the validation material is placed under `inst/validation` so it is preserved in the installed package — meaning the validation packet is present on the machine where the software actually runs.

The distinction to keep straight: **VALTOOLS IS FOR VALIDATING A PACKAGE YOU BUILD; RISKMETRIC AND VAL.METER ARE FOR ASSESSING PACKAGES YOU CONSUME.** Different problems, different tools, frequently conflated.

```
valtools::vt_create_package("adsl.tools")     # new package with validation scaffold
valtools::vt_use_validation()                 # or retrofit an existing one

valtools::vt_use_req("derive_trtdur.md")      # the requirement
valtools::vt_use_test_case("derive_trtdur.md")# the test case description
valtools::vt_use_test_code("derive_trtdur.R") # the executable test
valtools::vt_use_change_log()
valtools::vt_use_report()                     # renders the signed report
```

INTERVIEW TRAP. "Do we have to use valtools?" No — and it is worth saying that the framework matters more than the tool. What a sponsor needs is requirements, test cases traceable to those requirements, executed evidence, a change log, and an approved report. valtools automates producing those artefacts; a well-run team can produce the same artefacts in Word and a shared drive. The tool is not the compliance; the traceability is. Also worth flagging: none of the five senior R reqs surveyed named valtools, so treat it as evidence you know the landscape rather than as a box to tick.

Q25. Map IQ, OQ and PQ onto an R environment for me.

SHORT ANSWER. Installation qualification is objective evidence that R, its packages and the platform were installed as specified — versions, locations, checksums, permissions. Operational qualification is

evidence that the installed system functions as intended across its operating ranges — running R's own regression test suite, running your packages' test suites, verifying access control behaves. Performance qualification is evidence that it works for your intended use with your data and people, under your procedures.

FOR THE SAS PROGRAMMER. Structurally identical to what you did for a SAS installation, and you should say so — that recognition is reassuring to a hiring manager. Same three stages, different artefacts.

DETAIL. Concretely, what each produces:

IQ. An installed-inventory record: R version and build, operating system and version, the exact list of installed packages with versions, the library paths, file permissions, and the repository or snapshot the packages came from. `sessionInfo()` and `installed.packages()` generate most of it; capture it to a dated file, not to a screen.

OQ. R Core maintains a set of validation and regression tests that ship in the `tests` subdirectory of the source tarball, with a README describing how to run all of them or a subset. Running those against your installation and retaining the output is genuine, defensible operational qualification evidence for base R — and the R Foundation guidance explicitly frames them as available to end users and administrators for exactly that purpose. Add your internal packages' `R CMD check` results, and evidence that user access controls work as configured.

PQ. A dry run of a representative real workflow — read a real ADaM dataset, run a real derivation, produce a real table — against a known expected result, performed by the people who will actually use the system, following the SOP they will actually follow.

One piece of precision that will impress: **IQ/OQ/PQ IS INDUSTRY CONVENTION, LARGELY FROM GAMP, NOT FDA REGULATORY VOCABULARY.** FDA's own process validation guidance uses a lifecycle model with different stage names, and no regulation mandates the three-letter scheme for analysis software. It is the language most sponsors' SOPs use, so use it — but knowing it is convention rather than statute is exactly the distinction a senior person can draw.

```
# IQ evidence — capture, date, retain
writeLines(capture.output(sessionInfo()), "evidence/iq_sessioninfo_2026-08-02.txt")
write.csv(installed.packages()[, c("Package", "Version", "LibPath")],
          "evidence/iq_installed_packages_2026-08-02.csv", row.names = FALSE)
```

```
# OQ evidence for base R — R Core's own regression/validation suite
cd R-4.4.1/tests && make check
# or the fuller run
make check-all
```

INTERVIEW TRAP. "Who signs the OQ?" Not you, usually — and knowing that matters. Qualification records are approved by whoever your quality system names, typically QA plus the system owner, and the programmer executing the tests is not the approver. If you answer "I'd sign it", you have described a segregation-of-duties failure. The senior answer describes the roles, not just the activity.

Q26. What does 21 CFR Part 11 actually cover, and does it apply to your R analysis code?

SHORT ANSWER. Part 11 governs electronic records and electronic signatures — when FDA will accept them as equivalent to paper records and handwritten signatures. It covers things like system validation, audit trails, record retention and retrieval, access controls, and the components and controls around electronic signatures. It applies through predicate rules: a record is a Part 11 record

because some other regulation requires that record to be kept, and you keep it electronically. It is not a general "all software must be validated" rule.

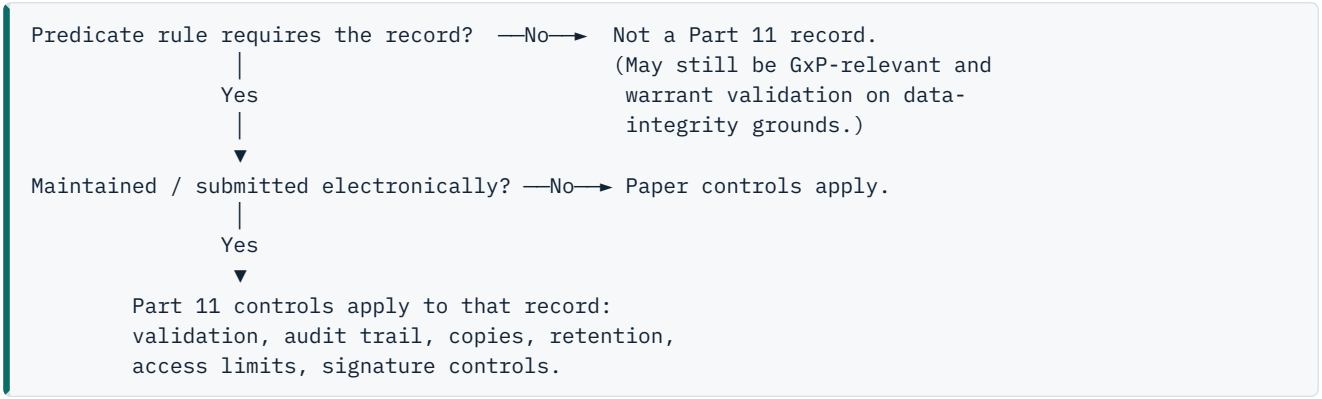
FOR THE SAS PROGRAMMER. You have almost certainly heard Part 11 invoked far more broadly than it actually reaches — it gets used as a shorthand for "compliance" in general. The predicate-rule mechanism is the correction, and it is the single most useful piece of regulatory precision in this whole module.

DETAIL. FDA's 2003 guidance, Part 11, Electronic Records; Electronic Signatures — Scope and Application, narrowed the enforcement position. The mechanism: if a predicate rule — such as 21 CFR Part 58 for GLP, Part 211 for CGMP, Part 312 for INDs, Part 314 for NDAs, Part 820 for devices — requires a record to be maintained, and you maintain it electronically, then Part 11 controls apply to that record. Same for signatures.

The passage that matters most here, and which the R Foundation guidance quotes: records submitted to FDA under predicate rules in electronic format are Part 11 records; however, if a record is not itself submitted but is used in generating a submission, it is not a Part 11 record unless it is otherwise required to be maintained under a predicate rule and is maintained electronically.

Read that carefully — it means statistical analysis software is generally not Part 11-mandated on its own. But two caveats you must give in the same breath. First, even where Part 11 does not apply, a system can still be GxP-relevant and warrant validation on data-integrity grounds. Second, when R runs as a component of a larger validated system — an eCTD workflow, a clinical data repository, a submission-ready output pipeline — parts of Part 11 do come into play for the records that system holds. And determining applicability remains the individual organisation's responsibility; that assessment is itself an expected deliverable.

Also name what Part 11 does not do: it does not say which statistical software to use, it does not require double programming, it does not require a particular test-coverage percentage, and it does not mention R, SAS, or any product.



INTERVIEW TRAP. The trap is the invitation to overclaim: "walk me through how your R environment is Part 11 compliant." A weak candidate lists features. The strong answer starts one step earlier: "First I'd establish which records in scope are Part 11 records via the predicate rule, because that determines what controls are actually required — then I'd map controls to those records." Starting with the applicability assessment rather than a feature list is the tell of someone who owns the framework rather than executing someone else's checklist.

Q27. Apply ALCOA+ to an R analysis.

SHORT ANSWER. ALCOA+ means data and records should be Attributable, Legible, Contemporaneous, Original and Accurate, plus Complete, Consistent, Enduring and Available. Applied to an R analysis it maps almost one-to-one onto version control, logging, environment capture and archiving — which is a useful way to explain to a sceptical auditor why the modern R toolchain is a data-integrity asset rather than a risk.

FOR THE SAS PROGRAMMER. Same principles you apply to CRFs, source data and study documentation, pointed at the analysis layer. Nothing new conceptually; the artefacts differ.

DETAIL. The mapping, which is worth being able to give fluently:

Attributable — every change has a named author via git commits and pull request approvals; runs are executed under named accounts, not a shared service account. Legible — code is readable and styled consistently, outputs are human-readable, logs are retained in plain text. Contemporaneous — commits, CI runs and analysis logs are timestamped at the moment of the activity, not reconstructed afterwards. Original — raw inputs are retained unmodified and read-only; derivations never overwrite source data. Accurate — verified by testing, independent programming and review. Complete — the whole chain is retained, including failed runs and the reasons they failed. Consistent — chronology is preserved and reproducible from the record. Enduring — retained in durable, controlled storage for the required period, not on someone's laptop. Available — retrievable on request throughout the retention period, which is where the container and package archiving discussion becomes concrete.

One point of precision worth having: ALCOA and ALCOA+ originate in data-integrity guidance in the GMP and GCP inspection space — FDA's data integrity guidance for drug CGMP, and the equivalent MHRA and WHO material — rather than in a rule specifically about statistical analysis. Applying them to analysis code is sensible extension and is now common expectation, but it is applied by analogy and by sponsor SOP.

```
# Contemporaneous + attributable + complete: capture the run, don't reconstruct it
run_log <- list(
  timestamp   = format(Sys.time(), "%Y-%m-%dT%H:%M:%S%z"),
  user        = Sys.info()[["user"]],
  code_commit = system("git rev-parse HEAD", intern = TRUE),
  r_version   = R.version.string,
  packages    = as.data.frame(installed.packages()[, c("Package", "Version")]),
  inputs      = tools::md5sum(list.files("data/raw", full.names = TRUE))
)
saveRDS(run_log, file.path("evidence", paste0("runlog_", Sys.Date(), ".rds")))
```

INTERVIEW TRAP. The follow-up is "Original — how do you prove the input data wasn't altered?" Checksums on the raw inputs at read time, recorded in the run log, plus read-only permissions on the raw data area, plus the fact that derived data is written to a separate location. If you answer only "we don't change it", you have described an intention rather than a control. Every ALCOA+ answer should name a control, not a habit.

Q28. What does traceability mean here, and how is it different from reproducibility?

SHORT ANSWER. Reproducibility means the same inputs and the same environment produce the same outputs again. Traceability means you can follow any number on any output backwards — to the program that produced it, to the dataset it read, to the specification that defines it, and forward again

to where it appears in the CSR. Reproducibility is a property of the machinery; traceability is a property of the documentation chain. You need both and they fail differently.

FOR THE SAS PROGRAMMER. Traceability is what your define.xml, ADRG and specification-to-program mapping already give you: a reviewer can pick a number in Table 14.1 and follow it to an ADaM variable, to its derivation, to the SDTM source. Reproducibility is whether re-running the program next year gives the same table. You can absolutely have one without the other, and both failures are real.

DETAIL. How each fails: reproducibility fails when the environment moves — a package updated, an R version changed, a random seed unset, a locale difference altering sort order. Traceability fails when documentation drifts — the spec was updated but the program was not, or the program was updated but the spec was not, or nobody can say which version of the package produced the archived table.

What links them in an R workflow is a run record. Each analysis run captures the code commit, the lockfile, the environment description, the input dataset identifiers and checksums, the outputs produced, the timestamp and the operator. With that record, a number on a table resolves to a specific commit and a specific environment, and that environment can be restored. Without it you have a reproducible pipeline that nobody can point at a specific past output.

Two traps in R specifically, worth naming unprompted because they show hands-on experience. Set the random seed explicitly for anything stochastic — multiple imputation, bootstrap, permutation tests — and record it, because "we used the default" is not reproducible across R versions; R has changed its default random number generator in the past. And be deliberate about sorting: locale affects string collation, so the same `arrange()` on the same data can order differently on two machines, which silently changes which record wins a "first non-missing" derivation. `sort(method = "radix")` gives a locale-independent ordering.

```
set.seed(20260802, kind = "Mersenne-Twister")   # record the seed AND the generator

# Locale-independent ordering — otherwise "first record per subject" can differ
dat[order(dat$USUBJID, dat$ADT, method = "radix"), ]
```

INTERVIEW TRAP. "Your table from last March needs regenerating. Walk me through it." The answer should be a chain, not a tool: find the run record for that output, get the code commit and the lockfile it names, restore that environment from the archived container image or lockfile, re-run against the archived input snapshot, and compare to the archived output within the tolerance your plan specifies. If any link in that chain is missing you should say which one and what you would do about it — an honest gap analysis is a better answer than a fluent description of a chain nobody actually maintains.

Q29. How does double programming work in R?

SHORT ANSWER. Exactly as it does in SAS, and that is the important point — it is a process control, not a language feature. Two programmers work independently from the same specification, produce the output separately, and the results are compared; differences are adjudicated against the spec, not against whichever program is prettier. The comparison tooling differs, and R gives you the option of a genuinely stronger form: the QC program can be a test suite rather than a second copy of the table.

FOR THE SAS PROGRAMMER. Your existing process transfers wholesale. The one thing to unlearn is `PROC COMPARE` as the only shape a comparison can take. In R the same job is done by `diffdf::diffdf()` for a

by-key dataset comparison with a PROC-COMPARE-flavoured report, `waldo::compare()` for a readable structural diff, `arsenal::comparedf()`, or `all.equal()` for a straight tolerance-based check.

DETAIL. Two structural points worth making in an interview.

First, independence is a property of people and information, not of code. If the QC programmer reads the production program before writing theirs, the exercise is a review dressed up as double programming, and correlated errors survive it. The control is procedural: independent programmers, same specification, no sight of each other's code until comparison.

Second, R lets you invert the QC deliverable in a way that is often more valuable. Instead of the QC programmer building a second table and comparing files, they write assertions derived from the specification — treatment group counts sum to the population total, the percentage denominator is the treated population not the enrolled population, no subject appears twice, the derived duration is inclusive of endpoints. That set of assertions is reusable across data cuts, runs automatically on every refresh, and catches the error class that a second identical implementation of the same misread specification will not. When the spec itself is ambiguous, two independent implementations can agree and both be wrong; an assertion set written from the spec surfaces the ambiguity instead.

```
# Classic double programming comparison
diffdf::diffdf(adsl_prod, adsl_qc, keys = "USUBJID", tolerance = 1e-8)

# Assertion-style QC, reusable on every data refresh
test_that("ADSL population flags are internally consistent", {
  expect_true(all(adsl$SAFFL[adsl$ITTFL == "Y"] == "Y"))
  expect_equal(sum(adsl$SAFFL == "Y"), nrow(adsl[!is.na(adsl$TRTSDT), ]))
  expect_equal(anyDuplicated(adsl$USUBJID), 0L)
})
```

INTERVIEW TRAP. "Does FDA require double programming?" No. No FDA regulation or guidance mandates double programming. ICH E9 says software used for data management and statistical analysis should be reliable and that documentation of appropriate testing procedures should be available; how a sponsor achieves that is its own decision. Double programming is an industry norm captured in sponsor SOPs — and saying that cleanly, then adding "which is why the interesting question is what my SOP should require and where", is a distinctly senior answer.

Q30. Independent programming or code review — how do you decide which a given output gets?

SHORT ANSWER. Risk-based, and documented in the validation or QC plan before the work starts. Independent programming for outputs where an undetected error has high consequence and where an independent implementation is genuinely feasible — primary and key secondary efficacy, safety outputs supporting labelling, derived analysis datasets. Code review plus targeted checks for lower-risk, high-volume, or structurally repetitive outputs. Never decided ad hoc by whoever has capacity.

FOR THE SAS PROGRAMMER. This is the QC tiering you already apply — you never double-programmed all 300 listings. What R changes is only the toolkit; the decision framework is identical and the interviewer wants to hear that you own the framework.

DETAIL. What each method actually catches is the useful lens. Independent programming catches implementation errors — a mistyped variable, a wrong join, a wrong denominator — because two people rarely make the same slip. It does not catch specification errors, since both programmers work from the same flawed spec. Code review catches specification misinterpretation, structural problems,

unhandled edge cases and poor error handling, because the reviewer is reading intent — but it is weaker at catching a subtle arithmetic error buried in correct-looking code.

That is why the strongest tier is not one or the other but a combination: independent programming plus review of the specification itself, plus an automated assertion suite that runs on every data refresh. And because R packages carry their own test suites, the reusable derivation logic can be validated once at package level and then does not need re-deriving per study, which lets you concentrate independent-programming effort where it actually pays.

The governance framing to lead with: the method is assigned per output class in a plan, approved before execution, and deviations are documented. If the interviewer is senior, that sentence is more valuable than any detail about tooling.

```
Output class                        Method
----------------------------------  ----------------------------------------
Primary / key secondary efficacy    Independent programming + spec review
                                    + assertion suite
Safety outputs supporting labelling Independent programming + assertion suite
ADaM analysis datasets              Independent programming (or validated
                                    package + independent check of results)
Standard, high-volume listings      Code review + automated structural checks
Exploratory / internal figures      Code review
```

INTERVIEW TRAP. "Your validated package already derives it — do you still double programme?" A genuinely good question and the answer is nuanced. If the package derivation carries requirements, tests, traceability and an approved validation report, you do not re-derive the function per study — that would be revalidating validated software. What you still verify per study is that it was applied correctly: right inputs, right parameters, right population, right specification. That distinction — validating the tool once versus verifying each use of it — is exactly the software-versus-output distinction from Q21, and connecting them shows the framework is joined up in your head.

Q31. A package you depend on releases a new version mid-study. What happens?

SHORT ANSWER. Nothing automatic — that is the entire point of pinning. The study is locked to its lockfile, so the new version does not enter the study environment until it is assessed and formally adopted. The new version is evaluated through change control: what changed, does anything the study uses behave differently, does it change any derived value, and is there a reason to adopt it at all mid-study.

FOR THE SAS PROGRAMMER. Identical logic to a SAS hotfix arriving during a study. You do not apply it because it exists; you assess it, and the default answer mid-study is "not now unless it fixes something that affects us".

DETAIL. The default should be explicit and defensible: **a study in flight does not change its environment without a documented reason.** The reasons that justify changing it are a defect affecting your results, a security issue in the environment, or a dependency that has become unavailable. Convenience and tidiness are not reasons.

When a change is justified, the assessment steps are concrete: read the NEWS file for the version range you are skipping, identify whether any changed function is one you call, re-run the full test and QC suite under the new version, and — the decisive check — re-run the study outputs under both versions and compare within the specified tolerance. If nothing moves, you have objective evidence

the change is inert for your use. If something moves, you now have a scoped, quantified impact statement rather than an opinion, which is what change control actually needs.

Retire the old environment only after the change is approved, and keep it archived — you may need it to reproduce an already-reported output.

```
# The study's environment is fixed by the lockfile
renv::restore()             # exactly the recorded versions
renv::status()              # flags any drift from the lockfile

# Controlled assessment of a proposed upgrade, in a branch, never in the study env
renv::install("dplyr@1.1.5")
renv::snapshot()            # produces a candidate lockfile for review
# then: re-run test suite, re-run study outputs, compare within tolerance
```

INTERVIEW TRAP. "What if the new version fixes a bug that affected your already-reported results?" Then this stops being a software question. It becomes a potential data-integrity and reporting issue: you assess which outputs are affected, quantify the impact with the statistician, raise it through the quality system, and the decision about whether anything reported needs correcting belongs to the study team and regulatory affairs, not to you. Candidates who keep answering in terms of version numbers here miss that the correct escalation is the answer.

Q32. You've written an internal package. What evidence does a sponsor expect before it can be used on a study?

SHORT ANSWER. A defined and approved set of artefacts: a validation plan stating scope and approach, a requirements specification, test cases traceable to those requirements, executed test evidence with results, the R CMD check output, a record of the environment it was qualified in, a change log, known issues, and an approved validation report signed per the SOP. Plus a defined process for how future versions are handled.

FOR THE SAS PROGRAMMER. This is the same packet you would assemble for a validated macro library — plan, requirements, test scripts, results, traceability matrix, summary report, signatures. Recognising that it is the same packet, with different file formats, is what makes this an easy transition to describe.

DETAIL. The traceability matrix is the artefact auditors reach for first, so build for it from the start: every requirement maps to at least one test case, every test case has executed evidence, and every executed test has a result and a date. valtools generates this from the repository structure via its scraping functions; you can equally maintain it manually, but manual matrices drift.

Content-level advice from experience: write requirements at the level of behaviour visible to the user, not implementation. "TRTDURD is derived as last dose date minus first dose date plus one day, inclusive of both endpoints, and is missing when either date is missing" is a testable requirement. "The function uses as.integer on a difftime" is an implementation note and will change without the behaviour changing.

Two things that are commonly missing and which reviewers notice. A **KNOWN ISSUES / LIMITATIONS** section — a validation report with no limitations reads as incomplete, not as flawless. And an explicit statement of the **environment** the package was qualified in: R version, platform, dependency versions. Software qualified in one environment is not automatically qualified in another, and stating the boundary is what makes the claim precise rather than open-ended.

```
Validation packet — adsl.tools v1.2.0
├── 01_validation_plan.md            scope, approach, roles, acceptance criteria
├── 02_requirements/                 one file per requirement, uniquely identified
├── 03_test_cases/                   test case per requirement, expected results
├── 04_test_evidence/
│     ├── testthat_results_2026-08-02.txt
│     ├── R_CMD_check_2026-08-02.txt
│     └── coverage_2026-08-02.html
├── 05_traceability_matrix.csv       requirement ID → test case ID → result
├── 06_environment.txt               R version, platform, dependency versions
├── 07_change_log.md
├── 08_known_issues.md
└── 09_validation_report.pdf         conclusion + approval signatures per SOP
```

INTERVIEW TRAP. "Who approves it?" Not the author. Approval routes through whoever your quality system names — typically the system or process owner plus QA, with the statistical lead for anything methodological. If you cannot name the approval roles you are describing a technical deliverable rather than a controlled one, and that is precisely the gap between "senior programmer" and "owns validation governance" that these roles are testing for.

Q33. An auditor sits down with you. What do they ask, and what do you say?

SHORT ANSWER. They will not ask about roxygen2. They ask governance questions: how do you know what software is installed, how did it get approved, who can change it, how do you know the outputs are right, and show me the evidence for this specific number. Every good answer names a control and points at a retained artefact.

FOR THE SAS PROGRAMMER. You have been in this room. The questions are the same ones; only the artefacts you point at change. Say that in the interview — it is reassuring and it is true.

DETAIL. The recurring questions and the shape of a good answer:

"What software was used to produce this table, and what version?" Point at the run record and the lockfile: R version, package versions, code commit, date, operator. This is also literally what the FDA's 2015 Statistical Software Clarifying Statement asks for — software fully documented including version and build.

"How did this package get approved for use?" Point at the risk assessment, the approval record, and the entry in the controlled repository. If the answer is "someone installed it", that is a finding.

"Who can change the production environment?" Point at access controls, the change-control procedure, and the deployment record. Note that the programmer being unable to change the production environment unilaterally is the desired answer.

"Show me that this derivation is correct." Point at the requirement, the test case, the executed evidence, and the independent programming or review record for the study output.

"What happens when a package is updated?" Point at the change-control SOP and an actual worked example.

"How would you reproduce this in five years?" Walk the chain from Q28, and be honest about its weakest link.

The meta-rule: **NEVER ANSWER AN AUDIT QUESTION WITH AN INTENTION.** "We always check that" is a finding waiting to happen. "That check is defined in SOP-xyz, executed at step 4, and here is the record from this study" is not.

```
Auditor question                    What you point at
----------------------------------  ----------------------------------------
Which software and version?         Run record + renv.lock + sessionInfo capture
How was the package approved?       Risk assessment + approval record + repo entry
Who can change production?          Access control config + change-control SOP
                                    + deployment record
Is this derivation correct?         Requirement → test case → executed evidence
                                    → independent programming / review record
What if a package updates?          Change-control SOP + a worked impact assessment
Reproduce it in five years?         Archived image + lockfile + code tag
                                    + input snapshot + retention record
```

INTERVIEW TRAP. The hardest one: "show me an example of when this process failed and what you did." Have a real answer ready. Auditors and interviewers both trust a candidate who can describe a genuine deviation, its root cause, the CAPA and whether the CAPA actually held, far more than one who claims nothing has ever gone wrong. A clean history is not credible; a well-handled failure is.

Q34. You join and inherit an R environment nobody has governed. What do you do first?

SHORT ANSWER. Inventory before change. Establish what is actually installed and used where, what evidence already exists, and where the real risk sits — then write the gap analysis and get agreement on priority. The instinct to start by installing renv and writing an SOP is the wrong order; you will govern the wrong things.

FOR THE SAS PROGRAMMER. Same as walking into an uncontrolled macro library. You do not start rewriting macros; you find out what exists, what is used on live studies, and what would actually hurt if it were wrong.

DETAIL. A defensible first-90-days sequence, and this is the answer these senior reqs are really probing for:

Inventory. What R versions and packages are installed, on which machines, and which studies depend on which. `installed.packages()` across environments is an afternoon's work and usually produces a surprise.

Criticality mapping. Which of those packages touch a primary endpoint, a submission deliverable, or a safety output. Most will not. Risk is concentrated, and finding out where is the highest-value thing you can do in month one.

Evidence gap analysis. For the critical few, what evidence exists today — tests, review records, approval trail — and what is missing. Write it down honestly. A documented gap with a plan is a defensible position; an undocumented gap is a finding.

Stop the bleeding. Two controls first, because they prevent the problem growing: a controlled repository so new packages enter through a decision, and version pinning per study so environments stop drifting. Both are cheap and both are immediately visible in an audit.

Then the framework: risk-assessment procedure, change control, qualification approach, roles and approvals — drafted with QA rather than presented to them.

The framing that separates senior from mid-level: **YOU ARE NOT BEING ASKED TO VALIDATE EVERYTHING; YOU ARE BEING ASKED TO KNOW WHAT NEEDS VALIDATING, TO WHAT DEPTH, AND TO BE ABLE TO DEFEND THAT DECISION.** That is risk-proportionate governance, and it is the same principle ICH E6(R3) applies to computerised systems — depth of validation matched to the consequence of failure, not skipped for low-risk systems and not uniform across all of them.

```
# Day one: find out what is actually there
inv <- as.data.frame(installed.packages()[, c("Package", "Version", "LibPath")])
inv$r_version <- R.version.string
write.csv(inv, "evidence/environment_inventory_2026-08-02.csv", row.names = FALSE)

# Then score the ones that matter, and file the result as a dated record
riskmetric::pkg_ref(critical_packages) |>
  riskmetric::pkg_assess() |>
  riskmetric::pkg_score()
```

INTERVIEW TRAP. "How long would it take to validate everything?" Refuse the premise, politely. Validating everything is neither achievable nor required, and proposing it signals you have not understood risk-proportionality. The answer: I would not attempt to; I would establish which uses are critical, apply depth proportionate to consequence, and document the rationale — and I would expect that to cover a small fraction of installed packages and the large majority of actual risk.

Q35. A CRAN package your study depends on gets archived. Now what?

SHORT ANSWER. For a study already pinned, nothing breaks immediately — the lockfile still names the version, and if you have retained the source tarball you can still install it. The exposure is future: no more fixes, no more compatibility updates, and eventually it will fail to build on a newer R. So it becomes a risk item to assess and plan out of, not an emergency.

FOR THE SAS PROGRAMMER. The nearest equivalent is a third-party component going end-of-life. You do not panic, you assess exposure, you plan migration, and you document the interim risk acceptance.

DETAIL. Why packages get archived matters: usually the maintainer stopped responding to CRAN's check failures, occasionally a licence or policy problem, sometimes deliberate deprecation. Archival says something about maintenance, not necessarily about correctness — a package can be archived and still compute correctly. That distinction is worth making, because the reflexive reaction is to treat archival as evidence of defect.

The response: confirm which studies use it and for what, check whether the functions you use are affected by whatever caused the archival, decide whether to vendor it (take the source under your own control and maintain it internally, which makes you the maintainer with all that implies), replace it, or accept the risk with a documented rationale and a review date. Record the decision.

The preventive control is the one to lead with: this is exactly why a risk assessment looks at maintenance activity, bug response and whether the project has a named maintainer and source control before you adopt a package. riskmetric's assessments cover precisely those indicators. Archival is usually the visible end of a decline that was measurable a year earlier.

```
# Retain what you depend on — do not rely on a public repository persisting
renv::snapshot()
download.packages(c("thatpackage"), destdir = "archive/tarballs",
                  repos = "https://packagemanager.posit.co/cran/2026-01-15",
                  type = "source")
```

INTERVIEW TRAP. "Isn't that a reason to avoid open source?" Do not get drawn into defending open source in general. The proportionate answer: every dependency is a supply-chain risk, including commercial ones — vendors discontinue products and change licensing too. The difference is that with open source you retain the source and can take over maintenance, which is an option a

commercial end-of-life does not give you. The mitigation in both cases is the same: assess maintenance risk before adopting, pin versions, retain artefacts, and review periodically.

Q36. Your Head of QA is sceptical about R. How do you make the case?

SHORT ANSWER. Not by advocating for R. By showing that the controls they already require can be met, mapping each one to a specific artefact, and being candid about where R is genuinely harder than SAS. Scepticism from QA is usually well-founded scepticism about governance, not about the language — so answer the governance question they are actually asking.

FOR THE SAS PROGRAMMER. You have watched people lose this argument by leading with capability. Nobody in QA cares that ggplot2 is nicer. They care who approved the software, who can change it, and how you know the numbers are right.

DETAIL. The case that works has four parts.

Establish the regulatory baseline. FDA's 2015 Statistical Software Clarifying Statement: the agency does not require any specific software for statistical analyses, and statistical software is not explicitly discussed in Title 21 — but the software used must be fully documented in the submission including version and build. So there is no rule to argue with; the question is entirely whether the sponsor can demonstrate fitness for purpose.

Show the industry precedent. The R Consortium Submissions Working Group has completed multiple pilot submissions through the FDA's eCTD gateway, progressively covering tables and figures, a Shiny application, ADaM datasets generated in R, WebAssembly and containers, and Dataset-JSON. This is not a theoretical route.

Map controls to artefacts. Take their existing SOP and show, control by control, which artefact satisfies it — approval routes to the repository decision record, change control to the git and change-log process, qualification to the IQ/OQ evidence, output verification to independent programming. QA reviews control coverage, so give them control coverage.

Name the hard parts yourself. R's dependency graphs are deeper and faster-moving than SAS's, so environment control needs more active management. The talent pool for validated R is thinner. Long-term reproducibility needs containers, which needs infrastructure support. Raising these before QA does is what converts you from advocate to trusted owner.

```
Their existing SOP control          The R artefact that satisfies it
--------------------------------    ----------------------------------------
Software approved before use        Risk assessment + repository approval record
Installation qualified              IQ inventory + R Core `make check` output
Change controlled                   Git history + PR approvals + NEWS + lockfile
Access restricted                   Repository permissions + branch protection
Outputs verified                    Independent programming / review records
Records retained and retrievable    Archived image, lockfile, tag, run record
```

INTERVIEW TRAP. "What if QA says no?" Do not say you would escalate or push harder. Say you would find out which control they believe cannot be met, because "no" is almost always a specific unmet control rather than a position — and if the gap is real, the right outcome may genuinely be to keep that workflow in SAS while building the capability elsewhere. Willingness to accept a no on evidence is a credibility signal; a programmer who treats QA as an obstacle is a liability in an inspection.

Q37. What does it mean to own validation governance rather than execute validation?

SHORT ANSWER. Executing means running the tests and producing the evidence someone else specified. Owning means deciding what needs validating and to what depth, writing or shaping the procedures that define it, being accountable for whether the framework holds under inspection, and being the person who says no when something has not met the bar. The technical skills are necessary but they are not what the accountability is made of.

FOR THE SAS PROGRAMMER. After eleven years you have almost certainly been doing pieces of this already — deciding QC tiering, writing conventions, being the person junior programmers ask. The interview task is to describe it in ownership language rather than activity language, because the job description is written in ownership language even when the interview questions sound technical.

DETAIL. What ownership looks like in practice, and these are the things to have concrete examples of:

Deciding scope. Which systems and packages are in scope for validation, at what depth, with the rationale recorded. Being able to defend both what you validated and what you deliberately did not.

Owning procedures. Drafting or revising the SOP and work instructions with QA rather than receiving them.

Being accountable at inspection. Being the person in the room who can explain the framework, produce the evidence, and answer for the gaps — including the ones you knowingly accepted.

Governing change. Owning the assessment of environment and package changes, and their impact across studies.

Building capability. Training programmers, defining what competence means for this work, and reviewing others' validation deliverables.

Saying no. Being willing to hold a release when the evidence is not there. This is the part that most distinguishes ownership, and it is worth naming explicitly in an interview.

The honest framing to close on: the tools in this module — packages, testthat, renv, riskmetric, valtools — are how you implement a framework. They are useful and they differentiate you. But none of the senior R reqs currently in market name them, because a hiring manager can teach a good programmer renv in a fortnight and cannot teach judgement about proportionate risk in a year. Lead with the judgement; keep the tools as evidence that you can actually build what you are proposing.

```
Executing validation                Owning validation governance
----------------------------------  ----------------------------------------
Runs the test scripts               Decides what gets tested and to what depth
Follows the SOP                     Writes and revises the SOP with QA
Produces the evidence               Defines what constitutes sufficient evidence
Reports an issue                    Owns the impact assessment and the CAPA
Uses the approved package list      Owns the process by which packages are approved
Answers a question at audit         Is accountable for the framework at inspection
```

INTERVIEW TRAP. The closing question is often "where do you see yourself in this role?" — and it is not small talk. If you answer in terms of learning R better you have described a mid-level trajectory. The answer that matches the seniority they are advertising: owning the validation framework for the R environment, making it inspection-ready, and building the team's capability to work within it — with the technical depth to build it myself rather than only to specify it. That sentence is what this whole module exists to make true.

Module 4 — Shiny, and Leading an R Transition

This module has two halves that get tested very differently. Part A is Shiny: the reactive model, modules, performance, testing, deployment, and what changes when a Shiny app influences a clinical or regulatory decision — and note that some senior job descriptions mention Shiny not as a build skill but as something you are expected to drive adoption of across studies, so you need both the hands-on answer and the portfolio answer. Part B is the leadership and behavioural layer that Principal, Associate Director and Director interviews actually spend most of their time on: migration strategy, sceptical stakeholders, vendor QC, hiring, and STAR stories you must fill with your own real experience and never invent.

Part A — Shiny

Section A1 — The Reactive Model

Q1. What is Shiny, and why is a Shiny app not just an R script that runs top to bottom?

SHORT ANSWER. Shiny is an R framework for building interactive web applications entirely in R — no HTML, CSS or JavaScript required, though you can use them. An app has a UI object describing what the browser shows and a server function describing how outputs are computed from inputs. It is not a top-to-bottom script because the server function does not "run" in order; it declares relationships, and Shiny re-runs only the pieces whose inputs changed.

FOR THE SAS PROGRAMMER. A SAS program is imperative: DATA step, PROC, PROC, done, in that order, once. Shiny is declarative and event-driven — closer to a spreadsheet than to a program. In Excel you do not tell cell C1 when to recompute; you write `=A1+B1` and Excel works out that changing A1 must update C1. Shiny is that, for R. Your job is to state the formulas correctly; Shiny owns the execution order.

DETAIL. Three pieces run at three different times, and confusing them is the number one beginner bug. Code outside `ui` and `server` (in `app.R` above them, or in `global.R`) runs **once per R process** — that is where you load packages and read reference data every user shares. Code at the top of the `server` function runs **once per user session** — that is where per-user state lives. Code inside a `reactive()` or `render*()` runs **whenever its dependencies invalidate** — potentially hundreds of times. Put a 40-second ADaM read at the top of `server` and every single user pays 40 seconds; put it above `server` and they pay it once, collectively.

```
library(shiny)

# runs ONCE per R process — shared by all users
adsl <- arrow::read_parquet("data/adsl.parquet")

ui <- fluidPage(
  titlePanel("ADSL explorer"),
  selectInput("trt", "Treatment", choices = sort(unique(adsl$TRT01A))),
  tableOutput("summary")
)

server <- function(input, output, session) {
  # runs ONCE per user session
  session_start <- Sys.time()

  # runs EVERY time input$trt changes
  output$summary <- renderTable({
    subset(adsl, TRT01A == input$trt) |>
      dplyr::summarise(N = dplyr::n(), MeanAge = mean(AGE, na.rm = TRUE))
  })
}

shinyApp(ui, server)
```

INTERVIEW TRAP. "Where do you put the code that reads the data?" is a disguised question about the three scopes. The trap answer is "at the top of the server function" — that is per-session and wastes memory and time. The follow-up is: "and what if the data changes while the app is running?" — then you need it inside a reactive with an explicit refresh trigger, not in global scope at all.

Q2. Walk me through the anatomy of a Shiny app — how do inputs and outputs actually connect?

SHORT ANSWER. Every input control in the UI has an `inputId`; the server reads it as `input$<id>`. Every output placeholder in the UI has an `outputId` and an `*Output()` function; the server assigns a matching `render*()` to `output$<id>`. The pairing is by string ID and by type — `plotOutput` pairs with `renderPlot`, `tableOutput` with `renderTable`, `DTOutput` with `renderDT`.

FOR THE SAS PROGRAMMER. Think of the ID as a macro variable name shared between two files. In SAS, if you `%let dsn = adsl;` in one place and reference `&dsn` in another, a typo gives you a silent or ugly failure. Same here: a mismatch between `plotOutput("ae_plot")` and `output$aeplot` produces a blank panel and no error message at all. Naming discipline matters more in Shiny than in SAS because there is no log line telling you the reference failed.

DETAIL. The commonly used pairs are worth memorising: `textOutput`/`renderText`, `verbatimTextOutput`/`renderPrint`, `tableOutput`/`renderTable`, `plotOutput`/`renderPlot`, `uiOutput`/`renderUI`, `downloadButton`/`downloadHandler`, plus package pairs `DT::DTOutput`/`DT::renderDT`, `plotly::plotlyOutput`/`plotly::renderPlotly`, `reactable::reactableOutput`/`reactable::renderReactable`. `input` is read-only inside the server — you cannot do `input$trt <- "Placebo"`; to change a control's value you call an `update*Input()` function. `output` is write-only — you cannot read `output$summary`.

```
ui <- fluidPage(
  sliderInput("bins", "Bins", min = 5, max = 50, value = 20),
  actionButton("reset", "Reset"),
  plotOutput("hist"),
  DT::DTOutput("tbl")
)

server <- function(input, output, session) {
  output$hist <- renderPlot(hist(adsl$AGE, breaks = input$bins))

  output$tbl <- DT::renderDT(
    DT::datatable(adsl, filter = "top", selection = "single"),
    server = TRUE          # paginate server-side; essential for big data
  )

  observeEvent(input$reset, {
    updateSliderInput(session, "bins", value = 20)   # never: input$bins <- 20
  })
}
```

INTERVIEW TRAP. "My output is blank and there is no error — what do you check?" Answer in order: ID typo between UI and server, wrong `render*` for the `*Output` type, the output is not visible on the currently selected tab (Shiny does not compute hidden outputs by default), or a `req()` upstream is silently cancelling. Candidates who only say "check the console" have not debugged a real app.

Q3. Explain the reactive graph, lazy evaluation, and invalidation.

SHORT ANSWER. Shiny builds a directed graph at runtime: inputs are sources, reactive expressions are intermediate nodes, and outputs and observers are endpoints. When an input changes, everything downstream is marked invalid rather than immediately recomputed. On the next flush, Shiny recomputes only the invalidated nodes that something actually needs — endpoints pull, sources push.

FOR THE SAS PROGRAMMER. It is the difference between rerunning a whole program and rerunning only the affected steps. Imagine a SAS batch where changing one macro parameter automatically figured out that steps 4, 7 and 9 depend on it and reran exactly those, in the right order, and left the other twelve alone. That dependency discovery is automatic — you never write the graph, Shiny observes which reactives you read and records the edge.

DETAIL. Two properties matter. **Lazy**: `reactive()` does nothing until something downstream reads it. If a reactive feeds only an output on a hidden tab, it never runs. **Cached**: within a flush cycle, a reactive computes at most once no matter how many consumers read it — this is why you put shared work in a `reactive()` instead of repeating the filter in three `render*` blocks. Dependencies are dynamic, not static: if your code has an `if` branch that reads `input$b` only sometimes, the edge to `input$b` exists only on the runs where that branch executed. That is a real source of "why did this not update" bugs.

```
server <- function(input, output, session) {
  # ONE node, computed at most once per flush, shared by both outputs
  filtered <- reactive({
    adae |>
      dplyr::filter(TRTA == input$trt, AESER == input$ser)
  })

  output$n_ae   <- renderText(nrow(filtered()))
  output$ae_tbl <- DT::renderDT(filtered())
  # If you instead repeated the dplyr::filter() in both render blocks,
  # it would run twice per change — and the two could silently diverge.
}
```

INTERVIEW TRAP. "Is `reactive()` eager or lazy?" Lazy. The follow-up trap is "and `observe()`?" — eager. That asymmetry is the single most tested reactivity fact. A second trap: candidates say "the reactive caches its value forever." It caches until invalidated, which is not the same thing, and it is per-session, not app-wide.

Q4. What is the difference between `reactive()` and `observe()`, and when do you use each?

SHORT ANSWER. `reactive()` returns a value, is lazy, and is cached — use it for computation. `observe()` returns nothing useful, is eager, and re-runs on every invalidation — use it for side effects like writing to a database, updating an input, or logging. If you want a value, use `reactive()`; if you want an action, use `observe()`.

FOR THE SAS PROGRAMMER. `reactive()` is a function that returns a dataset; `observe()` is a `%macro` you call for its effect — it writes a file, sends a mail, updates a tracking sheet. You would not use a data-returning function to write a log entry, and you would not use a side-effecting macro to produce a value you then read. Same discipline.

DETAIL. Call a reactive with parentheses — `filtered()`, not `filtered`. Forgetting the parentheses passes the function object around and produces baffling errors. The most common misuse is building state with `observe()` when a `reactive()` would do: writing `observe({ rv$data <- expensive() })` creates a hidden, eager, order-dependent path through your app that is far harder to test and to reason about than `data <- reactive(expensive())`. The rule of thumb senior interviewers want to hear: **prefer reactives; reach for observers only when the effect leaves the reactive world** — a file written, a database row inserted, an input updated, a notification shown.

```
server <- function(input, output, session) {
  # value: lazy, cached
  n_subjects <- reactive(dplyr::n_distinct(filtered()$USUBJID))

  # side effect: eager, no return value used
  observe({
    message("Subjects in view: ", n_subjects())   # logging
  })

  # side effect: keep a dependent control in sync
  observe({
    updateSelectInput(session, "site",
                      choices = sort(unique(filtered()$SITEID)))
  })
}
```

INTERVIEW TRAP. "What does `observe()` return?" It returns an observer object (invisibly) that you can `$destroy()` — but its value is not usable reactively. The deeper trap is the anti-pattern question: "here is an app that stores everything in `reactiveValues` updated by six observers — what would you change?" The expected answer is to convert as many as possible into reactives so the dependency graph is explicit rather than hidden in execution order.

Q5. `observeEvent()` versus `eventReactive()` — what is the difference?

SHORT ANSWER. Both fire only when a specified event expression changes, ignoring changes to anything else they read. `observeEvent()` performs a side effect; `eventReactive()` returns a value you call like a reactive. They are the "only run when the user clicks Go" pattern — one for actions, one for values.

FOR THE SAS PROGRAMMER. This is the batch-submit button. Without it, your app recomputes on every keystroke and slider nudge, which for a 3-million-row ADAE means a frozen browser. With it, the user sets five filters, clicks Apply, and then the work runs — exactly like assembling a parameter file and submitting the job once.

DETAIL. The signature order matters: `observeEvent(eventExpr, handlerExpr, ...)` and `eventReactive(eventExpr, valueExpr, ...)`. Anything read inside the **handler/value** expression does not create a reactive dependency — it is isolated automatically. That is the whole point. Useful arguments: `ignoreNULL = TRUE` (the default — do not fire on startup when the button is `NULL`/0), `ignoreInit = TRUE` (do not fire on the first run), `once = TRUE` (fire and then destroy). Modern Shiny also lets you write these as `reactive(...) |> bindEvent(input$go)` and `observe(...) |> bindEvent(input$go)`, which is the composable form and the one to mention if you want to sound current.

```
server <- function(input, output, session) {
  # value, computed only when Apply is clicked
  cohort <- eventReactive(input$apply, {
    adae |> dplyr::filter(TRTA %in% input$trt, AESEV %in% input$sev)
  })
  output$tbl <- DT::renderDT(cohort())

  # action, only when Export is clicked
  observeEvent(input$export, {
    readr::write_csv(cohort(), file.path(out_dir, "cohort.csv"))
    showNotification("Exported", type = "message")
  })

  # equivalent modern form
  cohort2 <- reactive({
    adae |> dplyr::filter(TRTA %in% input$trt)
  }) |> bindEvent(input$apply)
}
```

INTERVIEW TRAP. "Why did my `observeEvent` fire the moment the app started?" Because the event value changed from nothing to its initial value; fix with `ignoreInit = TRUE`. Second trap: putting the trigger inside the handler instead of the event slot — `observeEvent(input$go, { ... input$go ... })` still works but candidates often invert the arguments entirely and cannot explain which one is isolated.

Q6. `reactiveVal()` versus `reactiveValues()` versus `reactive()` — when do you use each?

SHORT ANSWER. `reactive()` is a computed value you cannot set. `reactiveVal()` is a single settable value: call it with no arguments to read, with one argument to write. `reactiveValues()` is a settable list-like container of several named values, read and written with `$`. Use `reactive()` whenever the value can be derived; use the settable ones only for genuine state that no formula can produce.

FOR THE SAS PROGRAMMER. `reactive()` is a derived variable in a DATA step — always recomputable from its inputs. `reactiveVal` / `reactiveValues` are more like a macro variable you `%let` at various points: powerful, but now the value depends on what happened in what order, which is exactly the debugging pain you already know from macro-heavy code.

DETAIL. `reactiveVal` reads as `x()` and writes as `x(new_value)` — a very common syntax error is `x <- new_value`, which silently destroys the reactive by rebinding the name. `reactiveValues` is created as `rv <- reactiveValues(a = 1, b = NULL)` and used as `rv$a`. Crucially, `reactiveValues` invalidates **per element**: a reader of `rv$a` is not invalidated by a change to `rv$b`. There is also `isolate()` for reading either without taking a dependency, and `reactiveValuesToList()` for snapshotting. The senior-level judgement is knowing that mutable reactive state is where large apps rot — every additional `reactiveValues` field is a piece of state whose correctness now depends on observer ordering.

```
server <- function(input, output, session) {
  # single settable value
  selected_id <- reactiveVal(NULL)
  observeEvent(input$tbl_rows_selected, {
    selected_id(adsl$USUBJID[input$tbl_rows_selected])   # WRITE
  })
  output$who <- renderText(selected_id())                # READ

  # several related pieces of state
  rv <- reactiveValues(queries = data.frame(), dirty = FALSE)
  observeEvent(input$flag, {
    rv$queries <- rbind(rv$queries,
                        data.frame(USUBJID = selected_id(),
                                   note = input$note,
                                   ts = Sys.time()))
    rv$dirty <- TRUE
  })
}
```

INTERVIEW TRAP. "What happens if you write `selected_id <- 'X01'` instead of `selected_id('X01')`?" You overwrite the reactive function with a character string; every later read fails with "attempt to apply non-function", and it happens far from the cause. The other trap: candidates claim `reactiveValues` invalidates everything that reads any element — it does not; invalidation is per-name.

Q7. What does `isolate()` do, and when do you genuinely need it?

SHORT ANSWER. `isolate()` reads a reactive value without creating a dependency on it, so a change to that value will not trigger a recompute. You need it when a block must use a value but must not be driven by it — the classic case being "recompute when the button is clicked, using whatever the filters say right now."

FOR THE SAS PROGRAMMER. It is the difference between a variable you monitor and a variable you merely look up. Your job triggers on the arrival of a new SDTM extract, but while it runs it also reads the config file — you do not want a config edit to relaunch the job. `isolate()` is that "read but do not trigger" distinction, made explicit.

DETAIL. In modern code you need `isolate()` less often than you used to, because `observeEvent()` / `eventReactive()` / `bindEvent()` isolate their handler bodies automatically — that covers most historical uses. The remaining legitimate uses are: reading a value inside an observer where you want only one of several reads to be a trigger; avoiding an infinite loop when an observer both reads and writes the same `reactiveValues` field; and capturing a snapshot for logging or for a download filename. Overusing `isolate()` is a smell — it means you are fighting the graph instead of shaping it.

```
server <- function(input, output, session) {
  rv <- reactiveValues(log = character())

  # triggers ONLY on input$run; reads the filters without depending on them
  observeEvent(input$run, {
    params <- list(trt = input$trt, sev = input$sev)   # already isolated here
    rv$log <- c(rv$log, paste(Sys.time(), "run with", params$trt))
  })

  # breaking a feedback loop: read rv$log without re-triggering on write
  observeEvent(input$clear, {
    old_n <- length(isolate(rv$log))
    rv$log <- character()
    showNotification(paste("Cleared", old_n, "entries"))
  })
}
```

INTERVIEW TRAP. "Show me an app that loops forever." An observer that reads `rv$x` and also assigns `rv$x` will re-trigger itself endlessly; the fix is `isolate()` on the read, or restructuring so the write happens in a different node. Interviewers also ask "could you use `observeEvent` instead of `isolate` here?" — usually yes, and saying so signals you know the modern idiom.

Q8. How do you stop an output from erroring before the user has made a selection? Explain `req()` and `validate()` / `need()`.

SHORT ANSWER. `req()` silently stops the current reactive if a required value is missing — no error shown, the output just stays blank. `validate(need(condition, "message"))` stops it too but shows a clean message to the user instead of a red error dump. `req()` is for "not ready yet"; `validate()` is for "you gave me something I cannot use, and here is why."

FOR THE SAS PROGRAMMER. `req()` is the early `%return` at the top of a macro when a required parameter is blank — quiet, expected, not an error condition. `validate(need(...))` is the `%put ERROR:` with a human-readable reason, except it appears in the user interface rather than the log. Both replace what a Shiny beginner does instead: let R throw, and show the user a stack trace containing internal object names.

DETAIL. `req()` treats `NULL`, `FALSE`, `""`, empty vectors, and unclicked action buttons (value 0) as "not truthy." `req(x, cancelOutput = TRUE)` is the important variant: it leaves the previous output on screen rather than blanking it, which stops a dashboard flickering to empty every time a filter is mid-edit. `validate()` takes one or more `need()` calls; `need(expr, message)` returns `NULL` when fine and the message when not. For file uploads and user-entered numbers, `validate()` is the right tool because the user needs to know what was wrong.

```
server <- function(input, output, session) {
  dat <- reactive({
    req(input$file)                              # nothing uploaded yet: stop quietly
    readr::read_csv(input$file$datapath, show_col_types = FALSE)
  })

  output$plot <- renderPlot({
    validate(
      need(nrow(dat()) > 0,               "The uploaded file has no rows."),
      need("AVAL" %in% names(dat()),      "Column AVAL is required."),
      need(input$cutoff > 0,              "Cut-off must be greater than zero.")
    )
    ggplot2::ggplot(dat(), ggplot2::aes(AVAL)) + ggplot2::geom_histogram()
  })

  output$tbl <- DT::renderDT({
    req(input$site, cancelOutput = TRUE)       # keep old table visible while switching
    dplyr::filter(dat(), SITEID == input$site)
  })
}
```

INTERVIEW TRAP. "What is the difference between `req(FALSE)` and `stop('...')`?" `req(FALSE)` raises a silent condition Shiny recognises and swallows; `stop()` surfaces as an error in the UI and in the log. A second trap: `req()` inside `observeEvent` is fine and common, but candidates often think `req()` only works in `render*`. And note `need()` must be inside `validate()` — calling `need()` alone does nothing.

Q9. How do you debug reactivity when an output updates at the wrong time, or not at all?

SHORT ANSWER. Use **reactlog**, which records the whole reactive graph and lets you step through invalidations visually. Enable it with `reactlog::reactlog_enable()` before running the app, press Ctrl+F3 (Cmd+F3 on Mac) in the browser to open the viewer while the app runs, or call `shiny::reactlogShow()` after it stops. Beyond that: `message()` / `cat()` inside reactives, `browser()` for interactive stepping, and `options(shiny.fullstacktrace = TRUE)` for deeper tracebacks.

FOR THE SAS PROGRAMMER. Reactlog is the closest thing Shiny has to `OPTIONS MPRINT SYMBOLGEN MLOGIC;` — it makes the invisible execution visible. In SAS you turn on the macro debugging options and read the log to see which macro resolved when. In Shiny you turn on reactlog and see the graph, node by node, with a slider through time showing what invalidated what.

DETAIL. Reactlog draws every reactive endpoint, expression and input as a node, colours nodes by state (running, invalidated, idle), and lets you scrub backwards through the session. It answers the two questions you cannot answer by reading code: "what actually depends on this?" and "why did this run four times?" Ctrl+Shift+F3 (Cmd+Shift+F3) marks a moment in time so you can find the exact user action later. It carries overhead, so it is a development tool — never leave it enabled in production. Complementary tools: `shiny::onStop()` and `session$onSessionEnded()` for teardown logging, and `shinyOptions(cache = ...)` diagnostics when you suspect caching rather than reactivity.

```
library(shiny)
library(reactlog)

reactlog_enable()            # sets the shiny.reactlog option

runApp("app")                # then press Ctrl+F3 in the browser

shiny::reactlogShow()        # or view the recorded log after the app stops
```

INTERVIEW TRAP. "How would you find out why a plot renders twice on startup?" A weak answer is "add print statements." The strong answer names reactlog, then explains the usual causes: an `observe` that updates an input which then re-triggers the plot, or `renderUI` recreating a control whose default value differs from the current one. Mentioning `freezeReactiveValue()` as the fix for the second case marks you out.

Q10. When do you use `renderUI()` versus `update*Input()` versus `conditionalPanel()`?

SHORT ANSWER. `update*Input()` changes an existing control's value or choices from the server — cheapest and the default choice. `renderUI()` / `uiOutput()` builds controls that do not exist yet, or whose number is unknown until runtime. `conditionalPanel()` shows and hides UI purely in the browser with a JavaScript condition, costing no server round trip.

FOR THE SAS PROGRAMMER. Updating an input is like changing the value list a prompt offers; `renderUI` is like generating the prompt itself with a macro because you did not know how many treatment arms the study would have. `conditionalPanel` is closer to a display-level `%IF` — nothing about the data changes, you are just hiding a panel.

DETAIL. Prefer `update*Input()` for cascading filters (choose a study, the site list narrows) because it preserves the control's identity, its current selection, and its client state. `renderUI` destroys and rebuilds the control, which resets its value and can cause a visible flicker and a spurious recompute — that is where `freezeReactiveValue(input, "id")` earns its place: it tells Shiny to treat the input as unavailable until the new UI arrives, suppressing the intermediate invalidation. `conditionalPanel` takes a JavaScript condition string using `input.foo`, not R syntax, and inside a module you must namespace it with `ns` via the `ns` argument.

```
server <- function(input, output, session) {
  # 1. cheapest: update an existing control
  observeEvent(input$study, {
    freezeReactiveValue(input, "site")
    updateSelectInput(session, "site",
      choices = sort(unique(adsl$SITEID[adsl$STUDYID == input$study])))
  })

  # 2. structure unknown until runtime
  output$arm_controls <- renderUI({
    arms <- sort(unique(adsl$TRT01A))
    lapply(arms, function(a)
      numericInput(paste0("target_", a), paste("Target N —", a), value = 50))
  })
}

ui <- fluidPage(
  selectInput("study", "Study", choices = c("A", "B")),
  selectInput("site", "Site", choices = NULL),
  uiOutput("arm_controls"),
  checkboxInput("adv", "Advanced options", FALSE),
  conditionalPanel("input.adv == true",          # JavaScript, evaluated in the browser
                   sliderInput("alpha", "Alpha", 0.01, 0.10, 0.05))
)
```

INTERVIEW TRAP. "Why did my selected site reset when I changed study?" Because `renderUI` rebuilt the control. The expected senior answer is to switch to `updateSelectInput` plus `freezeReactiveValue`. A second trap: `conditionalPanel` inside a module silently fails unless you pass `ns = session$ns` (or use `ns()` on the id in the condition string) — a classic namespacing bug.

Section A2 — Modules (the senior-level topic)

Q11. Why do Shiny modules exist, and when would you introduce them?

SHORT ANSWER. Because Shiny IDs are global: two copies of the same UI block collide on `inputId`, and a 3,000-line server function becomes untestable and unownable. Modules give each component its own ID namespace and its own server function, so a component can be instantiated many times, developed independently, unit-tested in isolation, and reused across apps. Introduce them as soon as an app has more than one screen or any repeated block — retrofitting later is painful.

FOR THE SAS PROGRAMMER. This is exactly the argument you already make for macros over copy-pasted code, plus one extra thing SAS macros do badly: scoping. A SAS macro variable is global unless you are disciplined with `%local`; a Shiny module namespace is enforced by the framework, so a module physically cannot reach the outside app's inputs. If you have ever debugged a macro variable clobbered three levels up the call stack, you understand why enforced namespacing is worth the ceremony.

DETAIL. A module is a pair of ordinary R functions: a UI function whose first argument is `id`, and a server function that wraps `moduleServer(id, function(input, output, session) { ... })`. The naming convention is `thingUI()` / `thingServer()`. Modules are the unit of ownership on a team — one person owns the AE module, another the lab module — and the unit of testing, because `testServer()` runs a module server without a browser. In a regulated setting they are also the unit of validation evidence: you can trace a requirement to a module and its tests. Frameworks that formalise this include **golem** (app-as-package) and **rhino** (opinionated structure, part of pharmaverse); both are worth naming.

```
# module UI ------------------------------------------------------------------
aeTableUI <- function(id) {
  ns <- NS(id)                              # ALWAYS the first line
  tagList(
    selectInput(ns("soc"), "System organ class", choices = NULL),
    DT::DTOutput(ns("tbl"))
  )
}

# module server --------------------------------------------------------------
aeTableServer <- function(id, ae_data) {    # ae_data is a reactive
  moduleServer(id, function(input, output, session) {
    observeEvent(ae_data(), {
      updateSelectInput(session, "soc", choices = sort(unique(ae_data()$AESOC)))
    })
    output$tbl <- DT::renderDT(
      dplyr::filter(ae_data(), AESOC == input$soc)
    )
  })
}

# app ------------------------------------------------------------------------
ui <- fluidPage(aeTableUI("serious"), aeTableUI("nonserious"))   # two instances, no clash

server <- function(input, output, session) {
  aeTableServer("serious",    reactive(dplyr::filter(adae, AESER == "Y")))
  aeTableServer("nonserious", reactive(dplyr::filter(adae, AESER == "N")))
}
```

INTERVIEW TRAP. "Can you just use a function that returns UI, without `NS()`?" You can, and it works until you instantiate it twice — then both copies write to the same `inputId` and behave as one control. Interviewers ask this to see whether you understand that the problem modules solve is ID collision, not code tidiness.

Q12. Walk me through `NS()` and `moduleServer()` — how does namespacing actually work?

SHORT ANSWER. `ns <- NS(id)` creates a function that prefixes any ID with `id` and a dash: `NS("ae")("tbl")` gives `"ae-tbl"`. Every input and output ID in the module UI must be wrapped in `ns()`. `moduleServer(id, function(input, output, session) {...})` creates namespace-aware `input`, `output` and `session` objects, so inside the module you use the bare names — `input$tbl`, not `input$"ae-tbl"`.

FOR THE SAS PROGRAMMER. Think of `ns()` as an automatic prefix on every symbol, like prefixing all your macro variables with the module name so nothing collides — except the framework applies it for you on the UI side and strips it for you on the server side, so you never write the prefix by hand in the server.

DETAIL. Three rules cover almost everything. First, `ns <- NS(id)` must be the first line of the UI function, and every `inputId` / `outputId` — including `brushId`, `clickId`, and IDs inside `conditionalPanel` conditions — goes through it. Second, the UI function returns a `tagList()`, not a `fluidPage()` or `pageWithSidebar()`, because the module is a fragment of a page, not a page. Third, inside `renderUI()` in a module server there is no `id` argument in scope, so you obtain the namespace with `ns <- session$ns`. For nesting, the outer module's UI calls the inner UI as `innerUI(ns("inner1"))`, and the outer server calls `innerServer("inner1")` with no `ns()` needed — the module server is already namespaced. Historically the server side used `callModule(myModule, "id")` with the module function taking `input, output, session` as its first arguments; `moduleServer()` replaced it in Shiny 1.5.0 and is what you should write today (note that `testServer()` only works with the modern style).

```
profileUI <- function(id) {
  ns <- NS(id)
  tagList(
    selectInput(ns("subject"), "Subject", choices = NULL),
    plotOutput(ns("plot"), brush = ns("plot_brush")),   # brush IDs need ns() too
    uiOutput(ns("extra"))
  )
}

profileServer <- function(id, subject_data) {
  moduleServer(id, function(input, output, session) {
    ns <- session$ns                                    # needed inside renderUI

    output$plot <- renderPlot(plot(subject_data()$AVAL))

    output$extra <- renderUI({
      tagList(
        checkboxInput(ns("show_ci"), "Show CI", TRUE),  # ns() again, from session$ns
        textOutput(ns("note"))
      )
    })

    observeEvent(input$plot_brush, {                    # bare name on the server side
      message("brushed in module ", id)
    })
  })
}
```

INTERVIEW TRAP. "Inside a module, how do you get the namespace for a control you build in `renderUI`?" The answer is `session$ns`; candidates who say "call `NS(id)` again" have not built one, because `id` is not in scope inside `moduleServer`'s inner function in the way they imagine. Second trap: forgetting `ns()` on one control out of eight — the app runs, that one control is simply dead, and there is no error.

Q13. How do you pass data into a module and get results back out?

SHORT ANSWER. Pass **reactives in** as ordinary function arguments — pass the reactive object itself, not its value, and call it inside the module. Return **reactives out** by having the module server return a reactive, or a named list of reactives, which the calling server assigns and uses. Never reach across the namespace boundary to another module's inputs.

FOR THE SAS PROGRAMMER. It is parameter passing with one twist. In SAS you pass a dataset name and the macro reads it once. Here you pass a live value that can change, so you pass the un-called reactive — `reactive(x)` not `x` — and the module re-runs whenever it changes. Returning a list of reactives is the equivalent of a macro that hands back several outputs, except each one stays live.

DETAIL. The convention is: arguments that never change go in as plain values; anything that can change goes in as a reactive and is called with `()` inside. The most common bug is calling the reactive at the call site — `aeServer("ae", filtered())` passes a frozen snapshot, and the module then never updates. Coming out, return `list(selected = reactive(input$row), n = reactive(nrow(dat())))`. Returning a `reactiveValues` object also works but is a weaker design: it lets the parent write into the module's state, which destroys the encapsulation you bought the module for. Two modules that must talk to each other should do so through the parent — module A returns a reactive, the parent passes it into module B — rather than through shared global state.

```
filterServer <- function(id, data) {
  moduleServer(id, function(input, output, session) {
    filtered <- reactive({
      req(input$trt)
      dplyr::filter(data(), TRTA %in% input$trt)
    })
    # return a NAMED LIST OF REACTIVES
    list(
      data = filtered,
      n    = reactive(dplyr::n_distinct(filtered()$USUBJID))
    )
  })
}

server <- function(input, output, session) {
  raw <- reactive(load_adae(input$study))

  flt <- filterServer("flt", raw)          # pass the reactive, NOT raw()

  tableServer("tbl", flt$data)             # parent wires module to module
  output$header <- renderText(paste("N =", flt$n()))
}
```

INTERVIEW TRAP. "What happens if you write `filterServer('flt', raw())`?" The module receives a static data frame; it renders correctly once and then never updates, with no error. This is the module bug, and being able to describe the symptom — "it works until you change the study, then nothing happens" — proves you have hit it. A follow-up: "how do two sibling modules communicate?" Correct

answer: through the parent, or via an explicitly shared reactive passed to both — not by peeking at `input` across namespaces, which the framework forbids anyway.

Q14. How do you manage state in a large Shiny app?

SHORT ANSWER. Push as much as possible into derived reactives so there is no state to manage; keep genuine state minimal, owned by the module it belongs to, and passed explicitly. For app-wide state, a single `reactiveValues` "store" created in the top-level server and passed into modules is a workable pattern, but every field you add is a piece of order-dependent behaviour. For state that must survive a session — saved queries, sign-offs, user annotations — use a database, not memory.

FOR THE SAS PROGRAMMER. You have seen the equivalent failure mode: a program driven by twenty global macro variables set by different includes, where correctness depends on execution order and nobody can safely change anything. The discipline is the same — derive what you can, keep the rest few, named, and owned.

DETAIL. Practical rules for a large app. (1) Default to reactives; a value that can be computed should never be stored. (2) Give each module its own state; the parent sees only what the module returns. (3) If you need cross-cutting state, create one `reactiveValues` in the top-level server and pass it down as an argument — it is explicit and greppable, unlike a global. (4) Use `session$userData` for genuinely per-session metadata such as the logged-in user. (5) Bookmarking (`enableBookmarking("url")` or `"server"`) gives you shareable, restorable app state, which in a clinical review context is extremely valuable because a reviewer can send a colleague the exact view. (6) Anything that constitutes a record — a query raised, a listing signed off — belongs in a database with a user and timestamp, never in a reactive value that dies when the browser closes.

```
# one explicit store, created once, passed down
server <- function(input, output, session) {
  store <- reactiveValues(
    user     = session$user %||% "local",   # Connect/Server populates session$user
    cohort   = NULL,
    queries  = data.frame()
  )

  cohortServer("cohort", store)             # modules receive it explicitly
  queryServer("query",  store, con = pool)  # persistence goes to the database

  # bookmarkable state so a reviewer can share the exact view
  onBookmark(function(state) state$values$cohort <- store$cohort)
  onRestore(function(state) store$cohort <- state$values$cohort)
}
```

INTERVIEW TRAP. "Where do you keep the list of subjects the reviewer has flagged?" If you answer "in a `reactiveValues`", the follow-up is "what happens when their browser crashes, or when a second reviewer needs to see it?" In a clinical review app the answer must be a database with user and timestamp — the in-memory answer is what separates a demo from a system somebody signs.

Section A3 — Performance

Q15. A user says the app is slow. How do you approach it?

SHORT ANSWER. Measure before changing anything. Distinguish three different slownesses: slow **startup**, slow **reaction** to a single user, and slow **under concurrency**. Profile with `profvis`, and in the overwhelming majority of clinical apps the answer is that you are moving or recomputing too much data, not that Shiny is slow.

FOR THE SAS PROGRAMMER. Identical instinct to tuning a long-running SAS job: you do not rewrite the DATA step first, you look at where the time goes, and it is nearly always I/O or a sort you did not need. A Shiny app reading a 2 GB SAS transport file at every filter change is the same mistake as re-reading a permanent dataset inside a loop.

DETAIL. Work through it in this order. **Data**: precompute. Do the heavy aggregation in a scheduled job and have the app read a small, analysis-ready parquet or a database view. Read columns you need, not the whole dataset. Push filtering and aggregation into the database with `dbplyr` rather than pulling everything into R. **Graph**: make sure shared work sits in one `reactive()` rather than being repeated in several `render*` blocks, and gate expensive work behind `bindEvent()` so it does not fire on every keystroke. **Caching**: `bindCache()` for repeated identical requests. **Rendering**: `DT::renderDT(..., server = TRUE)` so the browser is not sent a million rows; downsample points before plotting. **Concurrency**: one R process handles one computation at a time, so ten users sharing one process queue behind each other — the fix is more processes (Connect, Shiny Server Pro, or a container replica set), plus `ExtendedTask` so a long job does not block the session that started it. Load test with **shinyloadtest** and **shinycannon** before you promise a number.

```
# profile the whole app
profvis::profvis({
  shiny::runApp("app", display.mode = "normal")
})

# push work to the database rather than pulling data into R
library(dplyr)
ae_summary <- reactive({
  tbl(pool, "adae") |>
    filter(STUDYID == !!input$study, AESER == "Y") |>
    count(AESOC, TRTA) |>
    collect()                       # only the summary crosses the wire
})
```

INTERVIEW TRAP. "So Shiny does not scale?" The expected answer separates the language from the architecture: a single R process is single-threaded for computation, so scaling is a deployment question (process count, load balancing) plus a data question (precompute, database, cache). Candidates who blame Shiny rather than their data pipeline are revealing they have never profiled one.

Q16. What does `bindCache()` do, and what is the danger with it?

SHORT ANSWER. `bindCache()` attaches a cache to a reactive or a render function, keyed on expressions you supply: `reactive(...) |> bindCache(input$x, input$y)`. If the key has been seen before, the stored result is returned without recomputing. The danger is that **the key, not the body, defines correctness**

— if the computation depends on something not in the key, you will serve a stale or, worse, another user's result.

FOR THE SAS PROGRAMMER. It is a results cache keyed on the parameter set, like keeping a permanent dataset named after the parameters that produced it and reusing it if the parameters match. And it fails the same way: if the underlying source data was refreshed but your key only encodes the parameters, you happily return yesterday's numbers.

DETAIL. The signature is `bindCache(x, ..., cache = "app")`. The `...` are the key expressions and they are the reactive part — the value expression is no longer reactive, so anything it reads must also appear in the key. `cache = "app"` (the default) shares results across all sessions in the R process, which is the point but also the risk: with a key that does not fully determine the value, one user's cached result can be served to another. `cache = "session"` avoids cross-session leakage but loses the sharing benefit. You can pass a cache object directly — `cachem::cache_mem(max_size = 500e6)` or `cachem::cache_disk(...)`, the latter surviving restarts and shared across processes — usually configured globally with `shinyOptions(cache = ...)`. Defaults are 200 MB for app scope and 200 MB per session. Keep keys cheap and small: avoid hashing large data frames, and avoid environments or R6 objects because their serialisation may not reflect the changes you care about. If the value depends on a database table, put something like the table's maximum modification timestamp in the key. Pairing with `bindEvent()` after `bindCache()` makes the event the reactive trigger while the key expressions become pure key material.

```
# key must contain EVERYTHING the value depends on, including data freshness
ae_counts <- reactive({
  compute_ae_counts(input$study, input$trt, cut_date())
}) |>
  bindCache(input$study, input$trt, cut_date()) |>
  bindEvent(input$go)

# app-wide cache configuration
shinyOptions(cache = cachem::cache_disk("./cache-adae"))
```

INTERVIEW TRAP. "You cached on `input$study` but the function also reads `input$trt`. What happens?" Two failures at once: the reactive does not invalidate when `input$trt` changes, and it returns the wrong cached value. In a clinical app this is a data-integrity defect, not a performance bug — and saying that out loud is the answer they want. The other trap is `cache = "app"` in an app where users have different data-access rights: that is a PHI leak, and the correct answer is `cache = "session"` or a key that includes the user's permission scope.

Q17. What is `bindEvent()` and how does it relate to `eventReactive()` and `observeEvent()`?

SHORT ANSWER. `bindEvent()` is the composable, modern way to say "only run this when that changes." `reactive(expr) |> bindEvent(input$go)` is equivalent to `eventReactive(input$go, expr)`, and `observe(expr) |> bindEvent(input$go)` is equivalent to `observeEvent(input$go, expr)`. Because it is a separate step, it composes with `bindCache()` and reads more naturally in a pipeline.

FOR THE SAS PROGRAMMER. Same submit-button idea as before, expressed as a modifier rather than a different function. You write the computation once, then attach a trigger to it — like writing your macro and separately deciding what schedules it, instead of having two near-identical macros for "run on change" and "run on submit."

DETAIL. Key arguments mirror the older functions: `ignoreNULL` (default `TRUE`, so it does not fire before a button is clicked), `ignoreInit` (do not fire on the first flush), and `once`. Order matters when chaining: put `bindCache()` first and `bindEvent()` second. With that order the event expression becomes the reactive trigger and the cache key expressions become key material only — which is usually what you want for an expensive query behind an Apply button. The older `eventReactive` / `observeEvent` forms are not deprecated and remain extremely common, so know both; the reason to prefer `bindEvent` is composability, and the reason to keep using `observeEvent` is that most existing code is written that way.

```
server <- function(input, output, session) {
  # expensive query: runs only on Apply, cached across sessions by its key
  cohort <- reactive({
    query_adae(input$study, input$trt, input$sev)
  }) |>
    bindCache(input$study, input$trt, input$sev) |>
    bindEvent(input$apply, ignoreNULL = TRUE)

  output$tbl <- DT::renderDT(cohort())

  # a render function can be bound directly too
  output$plot <- renderPlot({
    plot_ae(cohort())
  }) |> bindEvent(input$apply)
}
```

INTERVIEW TRAP. "Does `bindEvent` replace `observeEvent`?" No — it is an alternative spelling, not a deprecation, and saying "deprecated" is a small but visible accuracy error. The real trap is chaining order: `bindEvent()` then `bindCache()` gives different semantics from `bindCache()` then `bindEvent()`, and only the latter is the "cache the result, trigger on the button" behaviour people usually intend.

Q18. A computation takes two minutes. How do you stop it freezing the app?

SHORT ANSWER. Use `ExtendedTask`, which has been the recommended approach since Shiny 1.8.1. It runs work outside the reactive graph, so the reactive graph returns to idle immediately and the user who launched the job can keep using the app. Classic promises-based async only unblocked other sessions; `ExtendedTask` unblocks the session that started the work too.

FOR THE SAS PROGRAMMER. It is submitting a job to a batch queue instead of running it in your interactive session. You get a ticket back, you carry on working, and the result appears when it is ready. The `input_task_button` even shows the "Processing..." state, which is your job-status line.

DETAIL. `ExtendedTask$new(func)` takes a function that **must return a promise** — typically `promises::future_promise({ ... })` with `future::plan(multisession)` configured at startup. Declare it at the top level of the server function so each user gets their own; declaring it at the top level of `app.R` shares one task across all visitors. You cannot read reactives inside the task function — Shiny raises an error — so pass values in as arguments, which are evaluated eagerly at `$invoke()` time and therefore safely snapshotted. `$invoke(...)` starts it, typically from an `observeEvent`. `$result()` is the bridge back into reactivity: read it in a `render*` and it behaves correctly for each state — silent when not yet invoked, an in-progress signal while running, the resolved value on success, and the original error re-raised on failure. Invoking again while running queues the call. `bslib::input_task_button("run", "Run")` plus `bslib::bind_task_button("run")` gives the button its busy state — note it works with

`input_task_button`, not with a plain `actionButton`, and binding the button does not itself trigger the task; you still need the `observeEvent`.

```
library(shiny); library(bslib); library(promises); library(future)
plan(multisession)

server <- function(input, output, session) {
  # top level of server(): one task per user session
  run_sim <- ExtendedTask$new(function(n, seed) {
    future_promise({
      set.seed(seed)
      bootstrap_hazard(n)         # the long job; no reactives readable in here
    })
  }) |> bind_task_button("run")

  observeEvent(input$run, {
    run_sim$invoke(input$n_boot, input$seed)   # values passed in, not read inside
  })

  output$result <- renderPlot(plot(run_sim$result()))
}

ui <- page_sidebar(
  sidebar = sidebar(
    numericInput("n_boot", "Bootstrap samples", 1000),
    numericInput("seed", "Seed", 42),
    input_task_button("run", "Run simulation")
  ),
  card(card_header("Result"), plotOutput("result"))
)
```

INTERVIEW TRAP. "What is wrong with the old promises-in-reactives approach?" It is harder to write and it does not make the app responsive for the user who triggered the work — only for other concurrent users. Second trap: reading `input$n_boot` inside the `future_promise` block. That is an error by design, because the input could change while the job runs; the point of passing arguments is to freeze the parameters at submit time — which is exactly the reproducibility argument you already make about capturing parameters in a job log.

Section A4 — User Interface

Q19. How would you lay out a modern Shiny dashboard today?

SHORT ANSWER. With **bslib**, which gives Bootstrap 5 theming and a set of layout components designed for dashboards: `page_sidebar()`, `page_navbar()`, `page_fillable()`, `card()`, `layout_columns()`, `layout_column_wrap()`, `sidebar()`, `value_box()`, and `bs_theme()` for corporate branding. It has largely replaced the older `fluidPage` plus `shinydashboard` approach for new work.

FOR THE SAS PROGRAMMER. Layout is the part of Shiny closest to designing an output shell. You are deciding what a reviewer sees first, what is one click away, and what never needs to be on screen. The same judgement you apply to a table shell — headline numbers at the top, detail below, footnotes accessible — is exactly what makes a dashboard usable.

DETAIL. `page_sidebar(title = , sidebar = sidebar(...), ...)` is the workhorse: filters on the left, content on the right, responsive on a laptop. `page_navbar()` with `nav_panel()` gives you tabs for distinct workflows (Overview, AE, Labs, Listings). `card()` with `card_header()` is the standard content

container; `layout_columns()` places cards on a 12-column grid; `value_box()` gives the headline metric tiles. `bs_theme(version = 5, bootswatch = "flatly")` or explicit colour and font arguments applies branding, and `bs_themer()` lets you tune it live during development. `input_dark_mode()` adds a light/dark toggle. Two accessibility points worth raising unprompted: do not encode meaning in colour alone, and make sure tables remain readable at 200% zoom — reviewers use these apps all day.

```
library(bslib)

ui <- page_navbar(
  title = "Study XYZ Safety Review",
  theme = bs_theme(version = 5, bootswatch = "flatly"),
  sidebar = sidebar(
    selectInput("study", "Study", choices = studies),
    checkboxGroupInput("trt", "Treatment", choices = arms),
    input_dark_mode()
  ),
  nav_panel("Overview",
    layout_columns(
      value_box("Subjects", textOutput("n_subj")),
      value_box("Serious AEs", textOutput("n_sae")),
      value_box("Deaths", textOutput("n_death"))
    ),
    card(card_header("AEs by system organ class"), plotOutput("soc_plot"))
  ),
  nav_panel("Adverse events", aeTableUI("ae")),
  nav_panel("Patient profile", profileUI("profile"))
)
```

INTERVIEW TRAP. "Have you used shinydashboard?" Fine to say yes, but the current-practice answer is bslib, and knowing that `page_sidebar` / `card` / `value_box` supersede `dashboardPage` / `box` / `valueBox` shows you have kept up. A follow-up worth pre-empting: "how do you make it match our corporate template?" — `bs_theme()` variables and a shared internal theming package, so every app in the portfolio looks the same without each team hand-rolling CSS.

Q20. How do you present tables and interactive graphics in Shiny?

SHORT ANSWER. For interactive tables, **DT** (a wrapper around DataTables) or **reactable**; for a static, publication-quality table, **gt** or **flextable**. For interactive graphics, **plotly**, usually via `ggplotly()` on an existing ggplot. The critical setting for large clinical data is server-side processing so the browser never receives the whole dataset.

FOR THE SAS PROGRAMMER. DT is a listing that the reviewer can sort, search and filter themselves — which removes a whole category of "can you re-run this sorted by site" requests. gt is closer to a production table with proper headers, spanners and footnotes. Knowing which one a given deliverable needs is the same judgement as choosing between PROC PRINT and PROC REPORT.

DETAIL. `DT::renderDT(datatable(x), server = TRUE)` paginates on the server; with `server = FALSE` the whole data frame is serialised to the browser, which will kill the session on a large ADAE. Row selection comes back as `input$<id>_rows_selected` (indices into the current data), and this is how you build drill-down: click a row in the AE table, show that subject's profile. `reactable` is the more modern alternative with nicer grouping and expandable rows. For plots, `plotly::ggplotly(p)` converts an existing ggplot, and `plotly::event_data("plotly_click", source = "...")` gives you click and selection events for drill-down. Native Shiny also supports `plotOutput(brush = , click = )` with `nearPoints()` / `brushedPoints()`, which is lighter than plotly when you only need brushing. For

downloads, `downloadHandler(filename = , content = )` — and in a clinical app the downloaded file should carry the filter parameters and a timestamp in its name or header, so nobody can produce an unattributable spreadsheet.

```
server <- function(input, output, session) {
  output$ae <- DT::renderDT(
    DT::datatable(ae_view(), filter = "top", selection = "single",
                  rownames = FALSE, options = list(pageLength = 25)),
    server = TRUE
  )

  # drill-down: table click drives the profile
  picked <- reactive({
    req(input$ae_rows_selected)
    ae_view()$USUBJID[input$ae_rows_selected]
  })

  output$profile <- plotly::renderPlotly({
    p <- ggplot2::ggplot(dplyr::filter(adlb, USUBJID == picked()),
                         ggplot2::aes(ADY, AVAL, colour = PARAMCD)) +
         ggplot2::geom_line()
    plotly::ggplotly(p)
  })

  output$dl <- downloadHandler(
    filename = function() sprintf("ae_%s_%s.csv", input$study,
                                  format(Sys.time(), "%Y%m%dT%H%M%S")),
    content  = function(file) readr::write_csv(ae_view(), file)
  )
}
```

INTERVIEW TRAP. "The app dies when we load the full AE dataset — why?" Almost always `server = FALSE` on a DT, or plotting a million points without downsampling. The senior follow-up: "who is allowed to download from this app, and how do you know what they downloaded?" — download events in a clinical app should be logged with user, timestamp and filter state, and if you have never thought about that, it shows.

Section A5 — Testing

Q21. How do you unit-test a Shiny module without a browser?

SHORT ANSWER. With `shiny::testServer()`, which runs a module's server function in a simulated session. You set inputs with `session$setInputs()`, then assert on reactives, outputs and the module's return value (`session$returned`). It is fast, needs no browser, and runs in CI — this is where the bulk of your Shiny testing should live.

FOR THE SAS PROGRAMMER. This is the equivalent of testing a macro by calling it with known parameters and comparing the resulting dataset against expected values, rather than eyeballing the final RTF. Same principle as your double-programming discipline: verify the computation independently of how it is displayed.

DETAIL. `testServer()` takes the server function, an `args` list for the module's non-reactive and reactive arguments, and an `expr` block that runs inside the module's environment — so you can refer to internal reactives by name, which is enormously useful. `session$setInputs(x = 1, y = 2)` simulates user input and flushes reactives. `session$returned` gives whatever the module server returned, so you

can test the contract between modules. Note it only works with modern `moduleServer()` style, not the old `callModule()` style. Reactives that depend on wall-clock time or on random numbers need seeding or injection to be testable — which is another argument for passing dependencies in as arguments rather than reading globals inside the module.

```
test_that("filter module returns only selected treatment", {
  testServer(
    filterServer,
    args = list(data = reactive(adae_test)),
    {
      session$setInputs(trt = "Drug A")

      expect_true(all(filtered()$TRTA == "Drug A"))          # internal reactive
      expect_equal(session$returned$n(),                     # returned contract
                   dplyr::n_distinct(
                     adae_test$USUBJID[adae_test$TRTA == "Drug A"]))

      session$setInputs(trt = "Placebo")
      expect_true(all(filtered()$TRTA == "Placebo"))
    }
  )
})
```

INTERVIEW TRAP. "Does `testServer()` test the UI?" No — it does not render anything, so a missing `ns()` in the UI function passes every `testServer()` test and still breaks the app. That is precisely why you need the browser-level layer as well, and volunteering that boundary is the answer that lands.

Q22. How do you test a whole Shiny app end to end?

SHORT ANSWER. With **shinytest2**, which drives a real headless Chrome via an `AppDriver` object, sets inputs, waits for the app to settle, and compares outputs or screenshots against stored snapshots. It is built on testthat, so it runs in the same suite and the same CI as your unit tests.

FOR THE SAS PROGRAMMER. It is regression testing of the whole deliverable: you record what the app produced under a known set of inputs, and every future change must either reproduce it or explicitly justify the difference. That is the same control as comparing this run's outputs against the last validated run before you release.

DETAIL. `AppDriver$new(app_dir, name = , height = , width = , seed = )` starts the app. `app$set_inputs(...)` sets values — and for a control inside a module the id is the **namespaced, hyphenated** form, `"moduleId-inputId"`, not dot notation. Always call `app$wait_for_idle()` after setting inputs, or assertions can run before reactives have flushed. `app$expect_values()` snapshots input, output and exported values as JSON; `app$expect_screenshot()` snapshots the visual, which is more brittle. `app$get_value(output = "id")` reads a single output for targeted assertions. Keep screenshot snapshots to a minimum and generate all snapshots on one platform — snapshots made on macOS and replayed on Linux CI will differ on font rendering. Prefer many fast `testServer()` tests plus a thin layer of end-to-end tests over the critical user journeys; end-to-end suites are slow and, if neglected, provide false assurance. Complementary tools worth naming: **shinyloadtest** with **shinycannon** for multi-user load, and **shinyValidator** (open-sourced from Novartis) which automates a quality audit of an app project. Neither shinytest2 nor teal appeared in the senior job descriptions I would expect you to be measured against, so treat them as differentiators you volunteer, not boxes you must tick.

```
library(shinytest2)

test_that("selecting a treatment updates the AE count", {
  app <- AppDriver$new(app_dir = "../../", name = "ae-flow",
                       height = 900, width = 1400)
  on.exit(app$stop())

  app$set_inputs(`ae-trt` = "Drug A")     # module input: "moduleId-inputId"
  app$wait_for_idle()

  expect_equal(app$get_value(output = "ae-n_ae"), "142")
  app$expect_values(output = c("ae-n_ae", "ae-tbl"))
})
```

INTERVIEW TRAP. "Why did the test pass locally and fail in CI?" Three usual causes: a snapshot generated on a different operating system, an assertion running before `wait_for_idle()`, or the app depending on the current date. Anything time-dependent must be injectable — freeze the data cut-off as a parameter rather than calling `Sys.Date()` inside the app. In a regulated setting that is not just a testing convenience, it is a reproducibility requirement.

Section A6 — Deployment and Security

Q23. Compare the ways of deploying a Shiny app, and how you handle credentials.

SHORT ANSWER. Four broad options: **shinyapps.io** (Posit's hosted service — fastest, but public cloud and so rarely acceptable for patient data), **Shiny Server Open Source** (self-hosted, free, but no authentication and one process per app), **Posit Connect** (the enterprise answer — authentication, access control by user and group, scheduling, multiple processes, environment variables, git-backed deployment), and **Docker/Kubernetes** (maximum control, maximum operational burden). Credentials never go in code: use environment variables read with `Sys.getenv()`, set through Connect's variable store or `.Renviron`, and keep `.Renviron` out of version control.

FOR THE SAS PROGRAMMER. This is the same decision you make about where a validated program runs. shinyapps.io is a personal workstation — fine for a prototype, unthinkable for unblinded data. Connect is the managed, access-controlled, backed-up server your organisation already insists on for SAS, with the same expectations: who can run it, who can see the output, who changed it, and can you restore it.

DETAIL. For a pharma environment the realistic answer is almost always Connect or a container platform behind the corporate identity provider. Connect gives you: single sign-on via LDAP/SAML/OIDC, per-content viewer and collaborator permissions, `session$user` and `session$groups` inside the app so you can enforce study-level access in R, environment variables held encrypted at the content level, multiple processes and worker limits so ten reviewers do not queue behind one, and deployment either from `rsconnect::deployApp()` or as **git-backed content**, where Connect watches a branch and redeploys on change. Note that git-backed content requires a `manifest.json` produced by `rsconnect::writeManifest()` and cannot then also be updated by `deployApp()` — pick one route. Reproducibility comes from **renv**: `renv.lock` is committed, `renv::restore()` rebuilds the exact library, and for validation you also want the R version and, ideally, a package repository snapshot (Posit Package Manager) pinned by date. Deployment records live in `rsconnect/` and should be committed so colleagues republish to the same target.

```
# secrets: never in the source, always from the environment
pool <- pool::dbPool(
  drv      = RPostgres::Postgres(),
  host     = Sys.getenv("DB_HOST"),
  dbname   = Sys.getenv("DB_NAME"),
  user     = Sys.getenv("DB_USER"),
  password = Sys.getenv("DB_PASS")      # set in Connect's env vars, not .R files
)
onStop(function() pool::poolClose(pool))

# authorisation inside the app, using the identity Connect provides
server <- function(input, output, session) {
  user   <- session$user
  groups <- session$groups
  allowed_studies <- studies_for(user, groups)
  observe({
    updateSelectInput(session, "study", choices = allowed_studies)
  })
}

# publishing
rsconnect::writeManifest()                      # for git-backed content
rsconnect::deployApp(appDir = ".", envVars = c("DB_HOST", "DB_NAME"))
```

INTERVIEW TRAP. "Can we just put it on shinyapps.io?" The expected answer is a firm no for anything with patient-level data, with reasons: data residency, no corporate SSO, and no validated-environment control. A second trap: candidates say "we use environment variables" but then `Sys.setenv()` inside the app or commit `.Renviron`. Also be ready for "how do you guarantee the app runs the same next year?" — renv lockfile plus pinned R version plus a dated repository snapshot, and a container image if the platform team supports it.

Q24. What are the main security risks in a Shiny app?

SHORT ANSWER. Treat every value in `input` as attacker-controlled, because a user can send arbitrary values regardless of what the widget offers. The big four are: SQL injection from pasting input into query strings, arbitrary code execution from `eval(parse(...))` on user text, unsafe file uploads, and exposing data the user is not entitled to see — which in clinical work means PHI/PII and unblinding.

FOR THE SAS PROGRAMMER. You already know that a macro variable used unchecked in a `WHERE` clause is dangerous — the same class of problem, but where the "user" is anyone with a browser and the network path to the server rather than a colleague on a controlled box. The mental shift is that the dropdown is only a suggestion; the server must revalidate everything.

DETAIL. **SQL**: never paste; use parameterised queries with `DBI::dbGetQuery(con, "... WHERE STUDYID = $1", params = list(input$study))` or `glue::glue_sql(..., .con = con)`. **Code execution**: never `eval(parse(text = input$x))`, and never let a user name a file, a column or a function that you then pass to `get()`, `system()` or `source()`. **Uploads**: check size limits (`options(shiny.maxRequestSize = ...)`), check the extension and the content, never trust `input$file$name` when building a path (path traversal), write to a temp directory, and never `source()` an uploaded file. **Authorisation**: the fact that a widget only offers three studies does not stop someone sending a fourth — filter server-side against the entitlements you looked up for `session$user`, on every request. **Data exposure**: `cache = "app"` can serve one user's result to another; error messages can leak table names and paths, so set `options(shiny.sanitize.errors = TRUE)` in production; downloads should be logged; and screenshots

of an app showing subject identifiers are a disclosure event just as much as a spreadsheet is. **Transport**: HTTPS always, session timeouts configured on the platform.

```
# parameterised query, not string interpolation
dat <- reactive({
  req(input$study %in% allowed_studies())        # re-check entitlement server-side
  DBI::dbGetQuery(
    pool,
    "SELECT USUBJID, AEDECOD, AESEV FROM adae WHERE STUDYID = $1",
    params = list(input$study)
  )
})

# upload handling
options(shiny.maxRequestSize = 30 * 1024^2)
observeEvent(input$file, {
  validate(need(tools::file_ext(input$file$name) == "csv", "CSV files only."))
  safe <- file.path(tempdir(), paste0(uuid::UUIDgenerate(), ".csv"))
  file.copy(input$file$datapath, safe)           # never build a path from the user's name
})

# production error hygiene
options(shiny.sanitize.errors = TRUE)
```

INTERVIEW TRAP. "The dropdown only contains studies the user can see, so is server-side checking really needed?" Yes — the client can send anything, and any answer that relies on the UI for security is wrong. The clinical-specific follow-up: "what is the worst-case failure of this app?" For a safety dashboard on a blinded study, the worst case is inadvertent unblinding, and being able to say that — and to describe the control that prevents it — is what a Director-level interviewer is listening for.

Section A7 — Shiny in a Clinical and Regulated Context

Q25. Is a Shiny app a validated system?

SHORT ANSWER. It depends entirely on what the app is used for. Validation attaches to intended use, not to the technology: an exploratory app that helps a medical monitor decide where to look needs far less than an app whose output goes into a submission or drives a safety decision. The right answer is a risk-based one — classify the use, then apply proportionate controls, and document the classification.

FOR THE SAS PROGRAMMER. You already live this distinction. An ad-hoc listing you produce to answer a data-management question is not a validated deliverable; the same numbers in a CSR table are. Nobody validates SAS itself — they validate the system and the process around a defined use. Shiny is no different, and saying so calmly is what defuses the question.

DETAIL. The credible framework to reference is the **R Validation Hub** (pharmaR), whose white paper "A Risk-based Approach for Assessing the Accuracy of R packages within a Validated Infrastructure" is the anchor document, supported by the `riskmetric` package and the `riskassessment` Shiny app for scoring package risk. The Hub is also building toward a public regulatory repository of transparently assessed packages. Real-world adoption is documented — a 2026 case study from SCHARP at Fred Hutch describes a historically SAS-based submitter building an R validation framework on the Hub's white paper and `riskmetric`, organised around study-specific purpose, development lifecycle, community usage and testing, and mandated only for high-profile trials because resources are finite. That last detail is worth quoting: risk-based means not everything gets the same treatment, and a leader who says that sounds like someone who has run the process rather than read about it. For the

app itself, the controls are ordinary software controls: requirements traceable to tests, version control, code review, `testServer()` and `shinytest2` suites in CI, a pinned environment via renv plus a dated repository snapshot, controlled deployment, and a documented release with IQ/OQ/PQ evidence proportionate to risk. Frameworks like **golem** and **rhino** exist precisely to make an app look like a package so this evidence is producible; `shinyValidator` automates part of the audit.

```
# the validation-relevant machinery is unremarkable software engineering
renv::snapshot()                      # exact package versions, committed
testthat::test_dir("tests/testthat")  # testServer + shinytest2 in CI
rsconnect::writeManifest()            # reproducible deployment bundle
sessionInfo()                         # captured with every release record
```

INTERVIEW TRAP. "So you would validate a Shiny app the same way you validate SAS?" The trap is answering yes or no. The correct shape is: the process is the same — intended use, risk assessment, requirements, testing, controlled release — while the artefacts differ, and the additional risk with an interactive app is that the user chooses the analysis at runtime, so your requirements must cover the range of user journeys, not one fixed output. A second trap is claiming R or Shiny is "not validated" as though it were a property of the software; that answer will lose the room.

Q26. What do 21 CFR Part 11 and audit trail requirements mean for a Shiny app?

SHORT ANSWER. If the app creates, modifies or signs electronic records that are required by a predicate rule, Part 11 applies: unique authenticated users, access controls, computer-generated time-stamped audit trails of creation, modification and deletion, and controls around electronic signatures. If it is a read-only view over records held elsewhere, most of the burden sits with the source system — but access control and traceability still matter.

FOR THE SAS PROGRAMMER. The question you already ask about any deliverable — "who produced this, from what data, when, and can we reproduce it?" — is the whole of Part 11 in one sentence. The new part is that an app lets a user act, so if a reviewer can flag a subject, raise a query or sign off a listing, you have just created an electronic record and inherited the obligations that go with it.

DETAIL. Practical acceptance criteria for the logging layer: the trail must be generated automatically by the system rather than maintained by users; time-stamped with a consistent synchronised clock, UTC recommended, precise enough to reconstruct event order; tied to an authenticated individual with username and role captured; and it must record old and new values for changes. It must not be alterable or suppressible without preserving the original. Audit trail scope has to be defined and verified during validation, and organisations typically hold a separate SOP for audit trail review and another for electronic signatures. Concretely in Shiny this means: identity from the platform (`session$user` from Connect, not a name the user types); an append-only table in a database, not a CSV on disk and not a `reactiveValues`; every state-changing action written with user, UTC timestamp, action, record key, old value and new value; and read events logged too where the record is sensitive. Note also EU Annex 11, which applies more broadly to computerised systems than Part 11's narrower record-and-signature focus, so a multinational sponsor is subject to both.

```
log_action <- function(pool, user, action, key, old = NA, new = NA) {
  DBI::dbExecute(
    pool,
    "INSERT INTO app_audit (ts_utc, app_user, action, record_key, old_value, new_value)
     VALUES ($1, $2, $3, $4, $5, $6)",
    params = list(format(Sys.time(), tz = "UTC", "%Y-%m-%dT%H:%M:%OS3Z"),
                  user, action, key, old, new)
  )
}

observeEvent(input$flag_subject, {
  log_action(pool, session$user, "FLAG_SUBJECT", picked(),
             old = NA, new = input$flag_reason)
})
```

INTERVIEW TRAP. "Where do you store the audit trail?" Answering "a log file" is a fail — files can be edited and are not tied to a record. Answering "we log with the `logger` package" describes application logging, which is useful for operations but is not a Part 11 audit trail. The distinction interviewers listen for is between diagnostic logging and regulated record-keeping: different destinations, different retention, different access control.

Q27. Give me a realistic example of a clinical Shiny app and what makes it hard.

SHORT ANSWER. Three that are worth describing: an **AE/safety review dashboard** for the medical monitor, a **patient profile viewer** that assembles everything known about one subject on one screen, and a **TLF explorer** that lets a statistician browse and compare outputs across a study. The hard part in each is never the Shiny code — it is data freshness, blinding, entitlement, and traceability of what the viewer actually saw.

FOR THE SAS PROGRAMMER. These are the deliverables you already produce, made interactive. The AE dashboard replaces a stack of listings a monitor reads once a fortnight. The profile viewer replaces the ten listings someone joins by hand when a serious AE comes in. The TLF explorer replaces emailing RTFs. Framing them that way in an interview shows you understand the business problem, which is what a Principal-level candidate is being assessed on.

DETAIL, USING THE AE DASHBOARD AS THE WORKED EXAMPLE. Data: ADSL plus ADAE, refreshed on a schedule from the validated ADaM environment, never re-derived in the app — the app reads analysis data, it does not create it, which keeps derivations under existing validation. Views: an overview with subject counts, SAE counts and deaths as value boxes; incidence by system organ class and preferred term by treatment arm; a drill-down from any cell to the subject listing, and from a subject to the profile view. Controls: blinding — if the study is blinded, treatment must be masked or the app restricted to the unblinded team by group membership, and this is the single most important design constraint. Entitlement: study-level access enforced server-side against the identity provider. Freshness: display the data cut-off date and extract timestamp prominently on every screen and in every download, because the most damaging failure mode is a monitor acting on data they believe is current. Traceability: filter state in every export, and downloads logged. Performance: precompute the incidence tables in the scheduled job so the app reads a small aggregate, with the subject-level pull happening only on drill-down. Testing: `testServer()` on the incidence calculations against an independently produced expected result — which is your double-programming discipline applied to an app. The **patient profile viewer** adds the challenge of aligning many domains on one time axis relative to first dose. The **TLF explorer** adds version control of outputs — which run produced this

table, against which data cut — and is often the easiest first app to justify because it saves the whole team time without touching a regulated derivation.

```
# the app reads validated analysis data; it does not derive it
adsl <- arrow::read_parquet(file.path(data_dir, "adsl.parquet"))
adae <- arrow::read_parquet(file.path(data_dir, "adae.parquet"))
cut  <- readRDS(file.path(data_dir, "meta.rds"))   # cut-off date, extract time, run id

ui <- page_sidebar(
  title = paste0("Safety review — data cut ", cut$cutoff, " (extract ", cut$extract, ")"),
  sidebar = sidebar(selectInput("study", "Study", choices = NULL)),
  layout_columns(value_box("Subjects", textOutput("n")),
                 value_box("SAEs",     textOutput("n_sae"))),
  card(card_header("Incidence by SOC"), plotOutput("soc"))
)
```

INTERVIEW TRAP. "What would you build first?" A candidate who names the most technically interesting app has missed the question. The strong answer picks the lowest-risk, highest-visibility target — usually a TLF explorer or a data-cleaning/listing review app — because it delivers value without touching a regulated derivation, and it earns the credibility you need before proposing anything that touches submission deliverables. The other trap: "how do you stop this app becoming a shadow analysis system?" — governance, a clear statement in the app that outputs are exploratory, and a hard rule that anything quoted externally comes from the validated pipeline.

Q28. Our job description says you will "drive adoption of R Shiny applications across studies." What does that mean to you?

SHORT ANSWER. It means the deliverable is not an app, it is a capability: a small catalogue of well-supported apps that many studies reuse, plus the platform, standards and support model that let study teams adopt them without each building their own. The failure mode to design against is fifty bespoke one-off apps with no owner, no tests, and no upgrade path.

FOR THE SAS PROGRAMMER. This is exactly the standard-macro-library problem you have solved before. One brilliant macro on one study creates no leverage; a maintained, documented, versioned library that every study uses creates enormous leverage — and the hard part was never the code, it was adoption, support and governance. Say that explicitly in the interview, because it maps your existing credibility onto unfamiliar technology.

DETAIL. A concrete plan you can articulate in two minutes. **One**: find the repeated pain — the thing every study team does by hand every fortnight — and build that, not the most impressive thing. **Two**: build it once, parameterised by study, so onboarding a new study is configuration rather than a fork. This is the single biggest architectural decision, and it is why modules matter: a study-agnostic app is a set of modules plus a study config. **Three**: put it on the enterprise platform with SSO and access control from day one, because an app on someone's laptop cannot be adopted. **Four**: define the support model before launch — who fixes it, what the SLA is, where bugs go — because the reason internal tools die is that the author moved teams. **Five**: measure adoption and time saved, in hours and in cycle time, and report it upward; leadership funds capabilities that have numbers attached. **Six**: govern the boundary — exploratory apps are free, anything touching a regulated deliverable enters the validation process described earlier. **Seven**: grow contributors, so app development is not a single point of failure. If asked what you would not do: I would not let every study team build its own app, and I would not start with an app that competes with the validated pipeline.

```
# the architectural move that makes adoption possible:
# one app, many studies, driven by configuration rather than forks
study_cfg <- yaml::read_yaml(file.path(cfg_dir, paste0(input$study, ".yml")))

server <- function(input, output, session) {
  aeTableServer("ae", reactive(load_domain(input$study, "adae")))
  labServer("lab",    reactive(load_domain(input$study, "adlb")),
            params = study_cfg$lab_params)
}
```

INTERVIEW TRAP. The trap is answering as a builder when the role is asking for an owner. “I would build a dashboard” is a Senior answer; “I would find the repeated pain, build it once as a configurable product, put it on the validated platform, resource the support model and report adoption” is a Principal/Director answer. Expect the follow-up “how do you handle a study team that insists on their own version?” — you accommodate genuine gaps by extending the shared app, and you refuse forks, because a fork is a maintenance liability with a study number on it.

Part B — Leadership & Behavioural

Section B1 — Strategy and Migration

Q29. How would you migrate a team from SAS to R?

SHORT ANSWER. Not all at once, and not by rewriting what already works. Phase it by risk: start where R adds value without touching a validated deliverable, build the infrastructure and the internal package library in parallel, run R alongside SAS on real studies before you rely on it, and convert submission-critical work only once the environment, the standards and the people are demonstrably ready. The goal is a team that can use both, not a team that has replaced one dependency with another.

FOR THE SAS PROGRAMMER. You have run transitions before — a new CDISC version, a new CRO, a new EDC, a new statistical computing environment. The pattern that works is the one you already use: pilot on something contained, prove it, standardise it, then scale, with an explicit rollback position at each stage. Say that out loud in the interview, because it tells the panel this is not your first migration, only your first migration to R.

DETAIL. A defensible four-phase plan. **Phase 0 — foundation, months 0 to 6.** Get the environment right first: a managed R installation, a snapshotted package repository (Posit Package Manager or equivalent) so builds are reproducible by date, renv in every project, git, and a review process. Without this, everything else produces work you cannot reproduce, and you will lose the QA argument permanently. **Phase 1 — low-risk, high-visibility.** Exploratory analysis, data review listings, feasibility and enrolment reporting, and Shiny apps for internal review. Nothing here is a submission deliverable, so nothing needs full validation, but people learn the language on real problems and produce visible wins. **Phase 2 — parallel run.** Pick two or three studies and produce a defined set of TLFs in R alongside the existing SAS production, comparing outputs. This is the phase that generates your evidence, your standards and your realistic effort estimates, and it is also where you find the genuinely hard bits — mixed models, survival edge cases, exact methods, legacy conventions. **Phase 3 — production.** Move new studies to R by default for the parts you have proven, keep SAS for the rest, and never migrate a study mid-flight. Throughout: an internal package library so 40 programmers are not writing 40 versions of the same function, a training path, and a metric you report upward. Explicitly name the things you would not migrate first: anything mid-study, anything for a submission inside twelve months, legacy studies with no future filings, and any area where R's implementation is not yet demonstrably equivalent.

INTERVIEW TRAP. The trap is presenting a technology plan when the panel wants a risk-and-people plan. Two follow-ups to be ready for: "what is your rollback if phase 2 fails?" — you keep SAS production running the whole time, which is the point of a parallel run and costs you effort rather than risk. And "how long?" — a genuinely honest range, three to five years to a mature dual capability for a mid-sized department, with visible value from year one. A candidate who says eighteen months to a full replacement is telling the panel they have not done this.

Q30. What would you deliberately not migrate to R, and when is SAS still the right choice?

SHORT ANSWER. Do not migrate anything mid-study, anything feeding a filing in the near term, legacy programs with no forward use, or areas where an R implementation is not yet demonstrably

equivalent to the SAS one. SAS remains the right choice where an existing validated pipeline works and the migration cost buys nothing, where the whole downstream chain including your CRO is SAS, and where a specific procedure has decades of regulatory familiarity and no settled R equivalent your statisticians will sign.

FOR THE SAS PROGRAMMER. This is the answer that establishes you as credible rather than evangelical. Every panel has met the person who wants to rewrite everything; almost none have met the person who can say clearly what should stay. Your eleven years are the reason you can say it and be believed.

DETAIL. The honest technical points. Mixed models are the standard example: `PROC MIXED` with a specific covariance structure and Kenward-Roger degrees of freedom has R analogues, but the burden of demonstrating equivalence for a primary endpoint is real work, and "we matched to the seventh decimal on three studies" is what it takes to retire that argument. Legacy conventions embedded in decades of macros — a particular rounding rule, a specific handling of missing at a visit — are not bugs to be fixed, they are consistency requirements across a programme. And the strategic point: the cost of a migration is not the rewriting, it is the revalidation, the retraining, the parallel running and the temporary drop in throughput while people are learning. If a study is filing in nine months, that cost lands exactly when you can least afford it. There is a governance answer too: a decision framework, published, that says which category of work goes to which language, so the choice is made once by policy rather than repeatedly by whoever is loudest.

INTERVIEW TRAP. "Are you saying R cannot do mixed models?" No — the accurate statement is that R can, and that the work is in demonstrating equivalence and getting the statisticians to sign, not in the capability. Be precise here; overclaiming and underclaiming are both damaging. The other trap is being asked this after you have given an enthusiastic migration answer, to see whether you will contradict yourself. Consistency between Q29 and Q30 is the actual test.

Q31. How would you convince a sceptical biostatistics or QA leader that R is acceptable?

SHORT ANSWER. Not with a language comparison. With evidence: a documented risk-based validation approach, reproducible environments, matched output on real studies, and precedent — including that the FDA has accepted submissions containing R analyses and a Shiny application through the R Consortium's R Submissions Working Group pilots. Then address their actual concern, which is almost never the language and almost always accountability if something goes wrong.

FOR THE SAS PROGRAMMER. You have sat on their side of the table. A QA leader is not asking "is R good"; they are asking "if an inspector asks me how this number was produced, can I answer, and who is accountable." Answer that question and the language question dissolves. If you have ever been the person who had to explain a discrepancy to an auditor, say so — it buys you more credibility than any technical argument.

DETAIL. Structure the case in four parts. **One, precedent.** The R Consortium R Submissions Working Group ran a series of pilots; Pilot 2 included a Shiny application as part of a publicly available submission package, with the FDA response letter received in September 2023. That is a concrete, checkable fact rather than an opinion, and it moves the conversation from "can we" to "how do we." Cite it precisely and do not overstate it — it establishes that the FDA accepted the submission package, not that any given app is pre-approved. **Two, framework.** The R Validation Hub's risk-based white paper, the `riskmetric` package and the `riskassessment` app give you an assessable, documented approach to package qualification rather than a hand-wave. Real sponsors are using it — the SCHARP case study from Fred Hutch, presented at PHUSE US Connect in 2026, describes a historically SAS-

based submitter doing exactly this, and notably mandating it only for high-profile trials because resources are finite. **Three, controls.** Show the machinery: version-controlled code, code review, unit tests running in CI, renv lockfiles, a dated package snapshot, controlled deployment, documented releases. Point out that this is more control than most legacy SAS macro libraries have, which is uncomfortable but usually true. **Four, evidence.** Parallel-run results on their own studies. Nothing persuades a QA leader like their own numbers reproduced. Finally, invite them in: ask them to define the acceptance criteria for phase 2 rather than presenting a finished plan. A sceptic who wrote the criteria becomes the person who signs them off.

INTERVIEW TRAP. "What if they still say no?" The wrong answer is to escalate. The right one is to find the smallest thing they will say yes to — usually a non-regulated internal deliverable — deliver it flawlessly, and come back with evidence. Also be ready for the sharpest version: "who is accountable if an R result is wrong?" The answer is you, exactly as you are accountable today for a SAS result, and the controls exist so that accountability is enforceable rather than nominal.

Q32. How do you handle the "R is free, so it must be unreliable" objection?

SHORT ANSWER. Separate price from support model. Base R and the recommended packages are maintained by the R Foundation and R Core with a formal release and testing process; the risk in the ecosystem is not the language, it is the variability in quality across 20,000 contributed packages — which is a real risk, and it is managed by curating and qualifying the subset you actually use. And commercial support exists: Posit sells supported R distributions, package managers and servers, so "free" is a choice, not a constraint.

FOR THE SAS PROGRAMMER. You can flip the argument gently. A paid licence has never been a validation artefact — you still validate your SAS programs, and the SAS installation is qualified by your own IQ/OQ process, not by the invoice. The question is what evidence you hold, and that question is answerable in both languages.

DETAIL. Make three moves. **First, name the real risk honestly** — package quality varies enormously, maintainers can abandon work, and an unpinned dependency can change under you. That honesty is what makes the rest credible. **Second, name the controls**: an internal curated repository so programmers install from an approved snapshot rather than from CRAN directly; risk scoring with `riskmetric` for anything used in a regulated deliverable; renv lockfiles so a study's environment is exactly reproducible years later; and a rule that no package enters the approved set without an owner. **Third, name the commercial option**: Posit Workbench, Posit Connect and Posit Package Manager come with vendor support and the qualification documentation an IT or QA function expects. Where organisations get burned is not choosing R, it is choosing R with no repository strategy — every laptop with a different package set and no lockfiles. Say that plainly; it shows you know where the bodies are buried.

INTERVIEW TRAP. "Who do we sue if R is wrong?" It is a real question asked semi-seriously, and the honest answer is that no vendor indemnifies you against a wrong analysis — read a SAS licence and you will find the same. The sponsor is accountable for the analysis in both worlds; what differs is the support contract, and one is available for R if the organisation wants it. Do not get drawn into defending open source ideologically; the panel is testing temperament as much as content.

Section B2 — Building the Capability

Q33. How would you build an internal R package library, and how do you get people to actually use it?

SHORT ANSWER. Build the smallest set of functions that solve problems every study has, package them properly with documentation and tests, distribute them through an internal repository, and treat adoption as a product problem: solve a real pain, make it easier than the alternative, support it, and version it so nobody is afraid to upgrade. Adoption fails far more often from lack of support and documentation than from lack of features.

FOR THE SAS PROGRAMMER. You have built or inherited a standard macro library, so you already know the two failure modes: nobody uses it because it does not fit real studies, or everybody forks it and now you have thirty variants. R gives you tools SAS never did — versioning, dependency declaration, automated tests, generated documentation, and an install command — which means the discipline you always wanted is finally enforceable.

DETAIL. WHAT TO BUILD. Start with the highest-frequency, lowest-variability tasks: reading and validating analysis-ready data, applying study standards to tables, common derivations, formatting and output conventions, environment and logging helpers. Do not start with a framework. **How to build it.** One package, or a small family, with roxygen documentation, `testthat` unit tests including edge cases for missing data and empty groups, semantic versioning, an owner, a public changelog and a deprecation policy. Also check whether pharmaverse already solves it — `admiral` for ADaM derivations, `rtables` / `tfrmt` / `gt` for outputs, `pharmaverseadam` for test data. Rebuilding what pharmaverse maintains is a real and common waste, and saying so shows external awareness. **How to distribute.** An internal repository so `install.packages()` just works, with the version pinned per study in `renv.lock`. **How to drive adoption.** Solve a pain people complain about, not one you find interesting. Pair with the first two teams and write their code with them. Publish worked examples using real study structures, not toy data. Make the support route obvious and answer fast — early responsiveness is what creates advocates. Never break the interface without a deprecation cycle, because one silent breakage will cost you two years of trust. And measure: how many studies, how many programmers, what it replaced.

INTERVIEW TRAP. "How do you stop teams forking it?" Partly governance, but mostly by making the shared version better and faster to adopt than a fork, and by extending it quickly when a team has a genuine gap. A fork is usually feedback that your package did not fit reality. The follow-up is resourcing: if you cannot name who maintains it and how much of their time is protected, the panel will conclude — correctly — that it will be abandoned within two years.

Q34. Your team is SAS-native. How do you hire and develop R programmers?

SHORT ANSWER. Do both, deliberately: hire a small number of genuinely strong R people to set standards and unblock others, and invest heavily in converting your existing SAS programmers, who already have the expensive knowledge — CDISC, therapeutic area, regulatory expectation, and judgement about clinical data. Language is the cheapest thing to teach; domain experience is not.

FOR THE SAS PROGRAMMER. This is the argument that protects your own team, and it is also the argument that gets you hired. An experienced clinical SAS programmer learning R is a far better investment

than an R expert learning what ADaM is and why a subject can have two baselines. Say that with conviction — it is true, and it signals you will not damage the team you inherit.

DETAIL. HIRING. For the first hires, prioritise people who have shipped production R in a regulated or otherwise disciplined setting: they know packages, tests, version control and reproducibility, not just syntax. Screen for that with a code-reading exercise rather than a whiteboard puzzle — hand them a messy function and ask what they would change and why. Look for evidence of collaboration: package contributions, code review history, documentation they wrote. Be realistic that in clinical R the market is small, so treat "grew from SAS" candidates as first-class. **Developing existing staff.** Training courses alone do not work; people forget within weeks without application. What works is real work with support: pair each learner with a mentor, give them a genuine but non-critical deliverable, review their code properly, and protect the time — if learning is expected on top of a full study load, it will not happen and people will conclude leadership is not serious. Create visible role models by having respected senior SAS programmers go first, because their peers will follow them and not follow a memo. **Retention risk.** The moment your people are good at R they become marketable; that is the cost of the strategy and the answer is career path, interesting work, and not making them the person who only ever does the boring conversions. **Structure.** Avoid a permanent two-tier split where "the R team" is separate; it creates resentment and a single point of failure. Aim for R capability distributed across study teams with a small central group owning standards and tooling.

INTERVIEW TRAP. "Would you replace SAS programmers with R programmers?" A brutal question sometimes asked to see how you treat people. The answer that is both decent and correct is no — you convert, because their domain knowledge is the scarce asset and the language is not. Expect the softer follow-up too: "what do you do with someone who will not learn R?" — you find out why, you make the path realistic, and you accept that in a multi-year transition there is legitimate SAS work; but you are honest with them about where the department is going rather than letting them find out too late.

Q35. You have a CRO delivering R-based deliverables. How do you manage and QC that?

SHORT ANSWER. Exactly as you would manage SAS deliverables, plus three additions: agree the environment up front, require the code and the lockfile as part of the deliverable, and verify you can reproduce their outputs in your environment before you accept them. If you cannot rebuild it, you have not received a deliverable, you have received a PDF.

FOR THE SAS PROGRAMMER. You already run vendor oversight — specifications, milestones, QC plans, review of their validation documentation, sampling their work rather than trusting it. All of that carries over. The new failure mode is environmental: a CRO on different package versions can produce genuinely different numbers from identical code, which has no real SAS equivalent and is the thing to write into the contract.

DETAIL. BEFORE THE WORK STARTS. Agree the R version, the package repository snapshot date, and the approved package list, and put it in the specification. Agree that deliverables include source code, `renv.lock`, and a run log with `sessionInfo()`. Agree coding standards and whether they use your internal packages or their own. Agree the QC approach — independent double programming, or a documented review with defined coverage — and require their evidence, not their assurance. **During.** Get code early and incrementally, not all at the end; review a sample properly rather than skimming everything. Watch for the classic vendor pattern of a heavily bespoke framework that only they can maintain, which is a lock-in risk regardless of language. **On receipt.** Reproduce: restore their lockfile,

run their code on your data, compare outputs bit for bit where you can. Then QC the analysis, not just the execution — independently derive a sample of key results, check population counts and denominators, check handling of missing data and of subjects who discontinued, and check the numbers reconcile across outputs. Anything that fails to reproduce is escalated as a finding, because it means the deliverable is not under control. **Relationship.** Set the expectation of transparency early and be consistent; vendors deliver to the standard you actually enforce on the first deliverable, not the one in the contract.

INTERVIEW TRAP. "They sent outputs that do not match when we re-run them. What do you do?" Do not start by assuming incompetence — establish first whether it is environment (different package versions), data (a different extract or cut-off), or code. That triage order is the answer they want, because it is the difference between a manageable process gap and a relationship problem. The follow-up: "what would you have done differently?" — pinned the environment in the contract, which is exactly why the lockfile clause exists.

Q36. How do you make the cost/benefit case for R to executive leadership?

SHORT ANSWER. Not on licence cost. Lead on capability, speed and risk: access to modern methods and visualisation, faster iteration and automation, a far larger talent pool, and reduced dependence on a single vendor. Be explicit about the investment required — infrastructure, training, parallel running, temporary throughput loss — because a business case that shows only upside is not believed by anyone who has funded a transition before.

FOR THE SAS PROGRAMMER. You have written business cases for resource, for tooling, for headcount. The same rules apply: quantify what you can, be honest about what you cannot, and name the risk of doing nothing. The distinctive risk here is talent — the pipeline of new graduates trained primarily in R and Python is a genuine long-term staffing problem, and it is the argument that lands hardest with executives because it is about the future rather than this quarter.

DETAIL. Frame it in four buckets. **Cost avoided** — licence spend is real but not the headline, and quoting it as the main benefit makes you look junior; mention it once and move on. **Capability** — modern methods available in R first, interactive apps that replace static deliverable churn, automation of repeated reporting, easier integration with data science and real-world evidence work. **Speed** — measure something concrete in the pilot, such as time from data cut to review-ready listings, and use your own numbers rather than vendor claims. **Risk** — vendor concentration on one side, and on the other the risk of transition, which you name and mitigate rather than hide. Then present the investment honestly: environment and platform, training time, the parallel-run overhead, and a realistic productivity dip while people learn. Give a staged funding ask tied to decision gates, so leadership can stop after phase 1 — that structure is far more fundable than a three-year all-or-nothing request. And offer the counter-case: where the organisation should keep SAS, which makes the whole paper credible.

INTERVIEW TRAP. "What is the return on investment?" Resist a fabricated percentage. The strong answer gives a measurable pilot metric, states the assumptions, and commits to reporting actuals at the gate. Made-up numbers get remembered and quoted back. The other trap: an executive asking "why now?" — the answer is talent pipeline and capability, not cost, and if you lead with cost you will be asked why you did not do it five years ago.

Section B3 — Behavioural (STAR)

> **READ THIS BEFORE THE NEXT FOUR QUESTIONS.** Everything below gives you structure and prompts only. Do not use a story you did not live. Interviewers at this level probe — they will ask what the other person said, what the data actually showed, what you would do differently — and an invented story collapses in about ninety seconds and ends the interview in practice if not in form. Spend an hour before the interview writing three or four real situations from your own eleven years into the skeletons below, in your own words, with real numbers where you remember them and "roughly" where you do not. If you genuinely have no example for a question, say so and offer the nearest real thing: "I have not had exactly that, but the closest was..." That answer is respected. A fabricated one is not.

Q37. How should you structure a behavioural answer at Principal/Director level?

SHORT ANSWER. STAR: Situation, Task, Action, Result — but weighted for seniority. Keep Situation and Task to about twenty percent, spend most of the answer on your Action, and always land a Result with something concrete. At this level, add a fifth element: what you changed afterwards so it could not recur, because leaders are assessed on systems, not heroics.

FOR THE SAS PROGRAMMER. Your instinct will be to explain the technical detail, because that is what earned you credit for eleven years. Resist it. The panel is listening for judgement, influence and consequence: who you had to bring with you, what you decided with incomplete information, what it cost, and what you institutionalised. The technical detail is context, not the answer.

DETAIL. THE SKELETON TO FILL. Situation — one or two sentences: study, phase, deliverable, timeline pressure, who was involved. Enough for the stakes to be obvious, no more. Task — what you specifically owned. Be careful with "we"; the panel needs to know what you did. Action — the bulk. Say what you decided, what alternatives you weighed, who you consulted, how you handled the people involved, and where you were uncertain. This is where seniority shows: a Principal answer includes the trade-off and the stakeholder, not just the fix. Result — quantify if you honestly can (days recovered, findings avoided, outputs corrected, people trained); if you cannot, say what changed and be straight about not having a number. Reflection — what you put in place afterwards: a check, an SOP change, a training item, a tooling change. **Delivery rules.** Two to three minutes; pause and let them ask. Name real constraints. Do not badmouth anyone — describe positions, not personalities, because how you speak about an absent colleague is itself being assessed. And be ready for "what would you do differently", which is asked of every senior candidate: have a genuine answer, because "nothing" reads as either untruthful or unreflective.

INTERVIEW TRAP. The two most common failures are burying the result and drifting into "we". A third, specific to technical leaders moving up: telling a story where you personally fixed everything. At Director level, a story where you fixed it alone is a story about a team you did not develop. Where it is true, show how you worked through others.

Q38. Tell me about a time you disagreed with a statistician.

SHORT ANSWER. This question is not about who was right. It is about whether you can hold a technical position without damaging a working relationship, and whether you know where the decision rights

actually sit. The strongest answers end with a decision that respected the statistician's ownership of the statistical method while your evidence changed what got decided.

FOR THE SAS PROGRAMMER. You will have several of these: a derivation you believed was wrong, an analysis population definition that did not match the protocol, a rounding or imputation convention, a change requested late that would have broken consistency with a previous submission. Pick one where you had data, not just an opinion, and where you can describe the other person's reasoning fairly.

DETAIL. PROMPTS TO FIND YOUR REAL EXAMPLE. When did you push back on a specification and turn out to be right? When did you push back and turn out to be wrong — that is often the better story, because it shows how you handle being wrong in public. When did a statistician want something the data could not support? When did you disagree about a protocol deviation, a censoring rule, or how to handle a subject's missing baseline? **Structure the answer.** State the disagreement neutrally and represent their position accurately and generously — the panel is watching for this. Say what evidence you brought: worked examples, the protocol or SAP text, output from both approaches side by side, precedent from a previous submission. Say how you raised it: privately first, in writing, with a proposal rather than an objection. Say who owned the decision — usually the statistician for the method, you for feasibility, implementation and quality — and say that you would have implemented their decision properly even if it had gone the other way. Land it: what was decided, what it prevented, and whether the relationship survived, which it must have if you are still describing them respectfully. **Reflection.** Did it lead to earlier specification review, a clearer sign-off gate, or a documented convention? Say so.

INTERVIEW TRAP. "What if they had overruled you and you still believed it was wrong?" You escalate on documented risk to data integrity or patient safety, through the defined route, once, calmly — and you do it in writing. On matters of statistical preference within a defensible range, you implement their decision and record the rationale. Getting that boundary right is the whole question. The other trap: a story where you "won" and the statistician looks foolish. That reads as someone who is difficult to work with.

Q39. Tell me about a time you caught a serious error.

SHORT ANSWER. Pick a real one, describe how you found it, what you did in the first hour, how you assessed the blast radius, how you communicated it upward and outward, and what control you put in place so it could not happen again. The panel cares more about your escalation behaviour and your systemic fix than about the clever catch.

FOR THE SAS PROGRAMMER. You have these. A wrong merge producing duplicate records, an analysis population that silently dropped subjects, a mis-specified visit window, a treatment mis-assignment, a QC comparison that passed because both programs shared the same wrong assumption, a number in a table that did not reconcile with a listing. Choose one that reached — or nearly reached — something external, because the stakes make the answer land.

DETAIL. STRUCTURE. How you found it — a reconciliation check, a review of an odd number, someone else's question you took seriously. If it was found by a check you had instituted, say so; that is a leadership point. First response — you verified it before raising it, quickly, and you did not sit on it. State the elapsed time honestly. Impact assessment — which outputs, which deliverables, which studies, what was already distributed, whether anything had gone to a regulator, a DSMB or a partner. That systematic blast-radius thinking is exactly what distinguishes a senior answer. Communication — who you told, how fast, and that you told them before you had the full answer rather than after.

Owning bad news early is the single most valued behaviour in this story. Correction — how the fix was made and re-verified, and by whom, independently. Prevention — the control you added: an automated reconciliation, a specification change, a QC step that checks the assumption rather than the arithmetic, a training point shared beyond your own team. **The strongest version** of this story involves an error you or your team made, not someone else's, because it lets you show accountability. If you have one you can tell without breaching confidentiality, it is worth more than a story about catching a vendor.

INTERVIEW TRAP. "How did you handle the person who made the mistake?" The expected answer separates the process failure from the individual, protects the person publicly, addresses it privately if needed, and fixes the system — because if a single mistake could produce that outcome, the control was missing, not just the care. Also: never name the sponsor, the compound or the study. Describing it generically is itself evidence you can be trusted with confidential work.

Q40. Tell me about a time you pushed back on a timeline.

SHORT ANSWER. Pick a case where you said a date was not achievable, said it early, and brought options rather than only a refusal. The assessment is whether you can deliver unwelcome news with evidence and alternatives, and whether you protect quality without being the person who always says no.

FOR THE SAS PROGRAMMER. Database lock moved forward, an SAP amendment arriving after production started, a CRO deliverable slipping and the downstream date not moving, extra analyses requested two weeks before a submission. Pick one where you can describe the actual arithmetic — programs, QC cycles, people, working days — because the credibility of this answer lives in the numbers.

DETAIL. STRUCTURE. The ask and why it mattered — represent the business pressure fairly; a filing date usually has real patient and commercial consequences and dismissing it makes you look parochial. Your analysis — show the work: the volume of outputs, the double-programming effort, the review cycles, the dependency you did not control. This is where you demonstrate estimation, which is a core skill at this level. How you raised it — early, with the evidence, to the right person, and not in a meeting where you ambushed them. The options you brought — usually some combination of scope reduction with the statistician's agreement, phasing so critical outputs land first, additional resource with a named cost and a realistic onboarding lag, or a specific, named risk you would accept knowingly. Presenting a knowingly accepted risk with mitigation is a senior move; refusing outright is not. What was decided and what happened — including if you were partly overruled and how you then made the compromise work. **Reflection** — did it change how estimates were made, when programming was brought into planning, or how change requests were controlled after study start? That is the real answer to "what did you learn".

INTERVIEW TRAP. "What if leadership said the date is fixed regardless?" You do not simply comply silently. You state clearly what will be reduced or what risk is being accepted, get that acknowledged in writing, and then execute the plan wholeheartedly. Quiet compliance followed by a missed date is the failure mode this question exists to detect. The other trap is a story where you were right and everyone else was unreasonable — include what the business was legitimately worried about.

Q41. You will be leading R programmers while being newer to R than they are. How do you handle that?

SHORT ANSWER. By being straight about it, and by being clear about what you do bring: eleven years of clinical judgement, regulatory experience, delivery leadership and vendor management, plus enough R to review work, ask good questions and make sound technical decisions with expert input. Pretending to expertise you do not have is the only genuinely fatal option.

FOR THE SAS PROGRAMMER. You have almost certainly led people who were better than you at something — a specific therapeutic area, a technology, a system you never used hands-on. The approach that worked then works now: respect the expertise, learn enough to be a competent reviewer, and add the value only you can add. What is different is that R is the visible currency of this team, so you cannot be indifferent to it; you have to be visibly learning.

DETAIL. Say three things in the interview. **First, honest positioning**: you are proficient in R and building depth, and you are not going to be the strongest R engineer in the room, nor should the Principal or Director be. **Second, what you do with that**: you set standards and hold people to them, you can read and review code and ask why a derivation handles missing data that way, you make the calls on architecture, validation, resourcing and vendor selection, and you deliberately create technical authority in the team rather than hoarding it — naming a technical lead, giving them decision rights, and backing them publicly. **Third, how you keep learning**: pairing, reviewing pull requests, building something real yourself. Doing a small internal Shiny app or an internal package end to end is disproportionately valuable, because it means you have felt the problems your team describes. If you have already done that, describe it concretely; it is the single most persuasive thing a career SAS leader can bring to an R interview. Finally, be ready to say what you would rely on others for — that is not weakness at this level, it is the definition of the job.

INTERVIEW TRAP. The trap is overclaiming, and it is usually sprung gently: a specific question about something you would only know from having built it, such as what breaks when you pass a called reactive into a module, or how you handled a package version conflict at deployment. Answer what you know, say plainly when you have not hit something, and describe how you would find out. Interviewers forgive gaps in a leader far more readily than they forgive bluffing — and the person you are bluffing to is frequently the technical lead who will report to you.

Q42. What should you ask the interviewer?

SHORT ANSWER. Ask questions that only matter to someone who intends to own the outcome: where the organisation actually is on R, who decides, what has already failed, how the role is measured, and what the first year looks like. Two or three good questions beat a list, and the best ones are follow-ups to something they said.

FOR THE SAS PROGRAMMER. Your questions are also evidence. Asking "what is your validation approach for R packages, and who owns the approved package list?" tells them more about your seniority than another answer would. Ask the things you would need to know on day one to decide whether the job is doable.

DETAIL. ON THE STATE OF PLAY. Where are you on the SAS-to-R journey today, honestly — pilot, parallel running, or production? What is already in R, and what is deliberately staying in SAS? Has an R initiative been tried before, and what happened? That last one is the highest-yield question in the set, because a failed prior attempt shapes everything. **On infrastructure and validation.** What is the

environment — Workbench, Connect, containers? Is there a curated internal repository and a package qualification process, or is that something this role builds? Who signs off that an R deliverable is fit for submission? **On the role.** What does success look like at twelve months, and who judges it? Is this role expected to build hands-on or to lead and govern — and if both, in what proportion? Where does it sit relative to biostatistics, data management and IT, and what decisions can it make without escalating? For roles mentioning Shiny adoption, ask directly whether they want apps built or an app capability established, because those are different jobs with different resourcing. **On people.** What is the current mix of SAS and R skills, and what is the plan for people who have not made the transition? How much protected time exists for upskilling? **On constraints.** What is the biggest obstacle right now — QA, infrastructure, skills, or study-team demand? What would you want the person in this role not to do in the first six months? **Closing.** Ask about their reservations: "Is there anything about my background that concerns you?" It is uncomfortable and it works, because it gives you the chance to address the SAS-to-R objection before it is decided in a room you are not in.

INTERVIEW TRAP. Asking nothing, or asking only about salary, remote working and headcount, signals you are evaluating the package rather than the problem. The other trap is asking a question the job description clearly answers, which reads as not having prepared. And listen to the answers — if they cannot say who owns package qualification, or the prior attempt failed for reasons nobody will name, that is information about whether the role can succeed, and a Director-level candidate is entitled to weigh it.

Module 5 — Portfolio scale: AI-assisted development, Git, reproducibility, standardization, and owning validation governance

Five live senior R programming reqs were checked on 2026-08-02 (ClinChoice, IQVIA, Novartis x2, GSK/ViiV): three of five explicitly require AI-enabled tooling — Novartis names “AI-enabled tools” in both reqs, GSK/ViiV asks outright for experience with “generative AI coding assistants (GitHub Copilot, Claude, Codex)” plus Agile delivery — four of five name Git, and four of five demand reproducible workflows and reusable standards across studies. At Principal/Associate Director level the reqs make you accountable for validation governance, not for executing QC, which is a different job from the one most candidates prepare for. Almost nobody walks in with a considered answer on any of this, so a calm, specific, regulator-literate position on AI and Git is the single cheapest way to sound senior in this market.

A note on currency: everything dated below was verified by search on 2026-08-02. Where the industry genuinely has not settled a question, this module says so — “the industry has not settled this, here is how I would decide in the absence of guidance” is a stronger interview answer than a confident wrong one, and a hiring manager at this level knows it.

Part A — AI-assisted development in a regulated environment

Section A1 — The core question and the accountability model

Q1. How would you use an AI coding assistant in a GxP environment?

SHORT ANSWER. I'd treat AI-generated code exactly as I treat code from a junior programmer or a contractor: it is unvalidated draft output until a qualified human reviews it, tests it, and signs it. The assistant changes how fast the first draft appears; it changes nothing about the control framework around it. The two things I'd control tightly are what data or IP goes into the tool, and the evidence that a human actually verified the output.

FOR THE SAS PROGRAMMER. You have already run this exact governance problem. When a CRO delivered you SAS code, you didn't accept it because the CRO is competent — you accepted it because your SOP required review, independent QC, and a signed record. The assistant is a new supplier of first drafts with no professional liability and no memory of your study. Everything you already do to a supplier's code, you do to its output.

DETAIL. A defensible position has four legs. First, **scope**: define where the tool may be used — exploratory work and non-GxP utilities freely; GxP deliverables (SDTM/ADaM derivations, TLFs, submission code) only under the full existing QC pathway. Second, **data boundary**: no patient-level data, no unblinded results, no sponsor-confidential specifications leave the approved tenancy. Third, **accountability**: a named human is the author of record on every commit, no exceptions and no "AI-authored" category in the system. Fourth, **evidence**: the review, the test cases, and the independent-programming comparison are recorded the same way they always were, so an inspector sees an unbroken control chain regardless of how the draft was produced.

The regulatory framing that supports this is helpful to have ready. FDA's January 2025 draft guidance, Considerations for the Use of Artificial Intelligence to Support Regulatory Decision-Making for Drug and Biological Products (docket FDA-2024-D-4689), sets a seven-step risk-based credibility framework anchored on "context of use" and borrowed from ASME V&V40 — and it explicitly carves out AI used for operational efficiency that does not affect patient safety, drug quality, or the reliability of study results. A coding assistant that drafts a program a human then validates sits in that carve-out. The code is still fully in scope of your normal computerised-system and GCP controls. That distinction — the tool is out of scope, the output is not — is the crux of the whole answer.

INTERVIEW TRAP. The follow-up is "so you'd let it write ADaM derivations?" Do not answer yes or no. Answer: it may draft them, and the draft has zero standing until independent programming and review agree. The trap is candidates who either ban it (sounds obsolete against a JD that requires it) or wave it through (sounds reckless). The senior answer is that the control point was never the keyboard, it was the review.

Q2. Who is accountable if AI-generated code produces a wrong number in a submission?

SHORT ANSWER. I am. Or whoever signed the deliverable is. Accountability is not delegable to a tool, and no regulator anywhere recognises a vendor as a responsible party for your analysis. The honest answer to "the AI got it wrong" is "my review missed it," and I would say that in a CAPA.

FOR THE SAS PROGRAMMER. Identical to a macro from a shared library producing a wrong result. Nobody ever accepted "the macro did it." The macro had an owner, a validation record, and a user who was responsible for applying it correctly. AI output has no owner and no validation record, so all of that responsibility lands on the user by default.

DETAIL. Say plainly that the accountability model is unchanged and then show you have thought about where it bites. AI shifts the failure mode: a junior programmer produces code that is wrong in visible, characteristic ways — an obviously mis-specified merge, a missing `by` group. A language model produces code that is wrong in plausible ways, because it is optimised to produce text that looks like correct code. That means the review has to be more substantive, not less: reviewing AI output by reading it for style is dangerous, because the style is always excellent.

The practical consequence is that you buy speed on the draft and you must spend some of it back on verification. A team that treats the assistant as pure throughput gain is where the incident comes from. I would say that out loud to a QA lead, because it is the true risk and it is the one nobody writes in the policy.

INTERVIEW TRAP. "Doesn't your vendor's terms give you some protection?" No. Commercial terms allocate commercial liability between two companies; they do not allocate regulatory accountability, which sits with the sponsor and, for a submission, with the named signatories. Anyone who thinks a licence agreement moves GxP accountability has misunderstood who the regulator talks to.

Q3. Is AI-generated code validated code?

SHORT ANSWER. No. It is unvalidated code, exactly like code I typed myself five minutes ago. That is the reassuring part of the answer: nothing about my validation obligations changes, because they never depended on who or what produced the first draft.

FOR THE SAS PROGRAMMER. You have never validated a programmer. You validated outputs, and you qualified the environment. A person's competence was managed by training records and review, not by validation. An assistant sits in exactly that slot — a producer of drafts, governed by review, not something you can validate once and then trust.

DETAIL. This is the single most useful sentence in the whole module and it is worth having word-perfect: the validation obligation attaches to the output and the environment, not to the author. From there everything follows. Your ADaM dataset is validated by independent programming and comparison against the specification. Your TLF is validated against the shell and the SAP. Your R packages are risk-assessed and qualified in the environment. None of those activities has an input field for "who wrote the draft."

Where AI would require validation in its own right is when the model itself becomes part of the analysis — a model imputing missing data, classifying adverse events, selecting endpoints, or producing a number that goes into a submission without a deterministic human-reproducible path. That is precisely what FDA's credibility framework is for, and that is a different conversation with a different risk profile. Being able to draw that line — assistant versus in-pipeline model — is what separates someone who has thought about this from someone repeating a vendor slide.

INTERVIEW TRAP. "Then why does your QA function care at all?" Because two things genuinely do change: what leaves the building (confidentiality), and whether reviewers stay honest when the draft looks good (human factors). Neither is a validation question, and saying so precisely is what makes the answer credible rather than glib.

Q4. How does this map onto our existing SOPs — do we need new ones?

SHORT ANSWER. Mostly you need one new policy and a few amendments, not a new SOP framework. The new policy is about acceptable use and the data boundary. The amendments are to the code review SOP, so a reviewer records that AI assistance was used and confirms substantive verification, and to your supplier/tool qualification process so the tool is on an approved list.

FOR THE SAS PROGRAMMER. Same shape as when your organisation first allowed code from an offshore team, or first allowed a new macro library. You didn't rewrite the QMS. You added an approved-supplier control and tightened the review step.

DETAIL. The minimum defensible set:

- **Acceptable use policy** — named approved tools and tenancies, prohibited inputs, prohibited use cases (never for statistical method selection, never as the sole source of a derivation rationale), and a requirement that the human is author of record.
- **Code review SOP amendment** — a checkbox recording AI assistance, and explicit reviewer confirmation that logic was verified against the specification rather than read for plausibility.
- **Tool qualification** — the assistant is a supplier. Assess it like one: contract terms, data handling, retention, tenancy, exit.
- **Training** — a short, mandatory module on prompt discipline and verification. The EU AI Act's AI-literacy duty (Article 4, applicable since 2 February 2025) gives you a compliance reason to fund it, if you need one.

Deliberately keep this light. Governance that is heavier than the risk gets routed around, and a policy people bypass is worse than no policy because it destroys your evidence trail.

INTERVIEW TRAP. Someone will ask whether you need to disclose AI assistance to a regulator in the submission. As of 2026-08-02, no guidance requires disclosure of coding-assistant use for programs that a human validated, and **the industry has not settled this**. My position: capture it internally so you can answer if asked, do not volunteer a new category of statement into a submission that no guidance asks for, and revisit if FDA finalises the 2025 draft or if ICH picks it up.

Q5. What does GAMP 5 second edition say about this, and does it apply?

SHORT ANSWER. GAMP 5 second edition (2022) added Appendix D11 on AI/ML and formalised the critical-thinking, risk-based approach — but it is aimed at computerised systems performing GxP functions, not at a developer's coding assistant. ISPE followed it with a standalone GAMP Guide: Artificial Intelligence in July 2025, roughly 290 pages, aligned to ISO/IEC 42001:2023. I'd use both as the framing language, while being clear that a coding assistant is a tool used by a person, not a system performing a GxP function.

FOR THE SAS PROGRAMMER. GAMP category thinking is the same instinct you already apply when you ask whether a piece of software needs IQ/OQ/PQ or is just an editor. Your text editor was never validated. Neither is a coding assistant, on the same logic.

DETAIL. The useful GAMP 5 second edition themes for this conversation are critical thinking over documentation volume, leveraging supplier evidence rather than duplicating it, and scaling rigour to risk. Applied here: you do not run OQ/PQ on a coding assistant; you assess the supplier, you constrain the inputs, and you rely on the existing verification of the output.

Worth knowing as adjacent context, because it shows you read regulators and not just vendors: draft EU GMP / PIC/S **ANNEX 22 ON ARTIFICIAL INTELLIGENCE** was published 7 July 2025 alongside a revised Annex 11 and Chapter 4; consultation closed 7 October 2025 with roughly 1,300 comments, and a final text is expected around end-2026. Its most quoted position is that critical GMP applications should use static, deterministic models, and that generative AI and LLMs are explicitly excluded from critical applications while permitted for non-critical uses such as documentation and training, with human oversight. Note carefully: Annex 22 is **GMP**, so it does not directly govern clinical statistical programming. But it is the clearest published statement of how European inspectors are thinking about generative AI, and "LLMs are acceptable for non-critical work under human oversight" is a very good sentence to be able to attribute to a regulator rather than to yourself.

INTERVIEW TRAP. Do not claim Annex 22 governs your ADaM programming — a good interviewer knows it is manufacturing, and overclaiming a citation is exactly the regulatory hand-waving that loses you the room. Cite it as directional evidence, label it as draft, and label it as GMP.

Section A2 — Confidentiality and the data boundary

Q6. What must never be pasted into a third-party model?

SHORT ANSWER. Patient-level data of any kind, unblinded results, treatment assignments, and sponsor-confidential material — protocol sections, SAPs, specifications, anything under an ongoing CDA. The rule I'd give a team is a sentence, not a matrix: if you would not email it to someone outside the company, it does not go in the prompt.

FOR THE SAS PROGRAMMER. You already know this instinct as the firewall around unblinded data. Everyone in this industry has watched an unblinding scare. A prompt box is a channel out of the building that has no DLP on it by default and no audit trail you control.

DETAIL. The categories, in order of how badly they end:

- **Patient-level data.** Even "just a few rows to debug." Even with the subject ID removed — dates plus site plus demographics re-identify readily, and HIPAA Safe Harbor requires removal of eighteen identifier types, which ad hoc de-identification never achieves.
- **Unblinded results.** Treatment codes, unblinded summaries, anything that reveals arm. The confidentiality risk here is not privacy, it's trial integrity.
- **Sponsor-confidential specs.** Protocols, SAPs, ADaM specs, shells. For a CRO this is contractual exposure; the CDA does not have a carve-out for convenience.
- **Credentials and connection strings.** Obvious, and still the most common real-world leak.

The rule scales badly if you write it as a taxonomy, so write it as a default-deny with a short allowlist: public documentation, CDISC standards, synthetic or published example data, your own utility code that contains no study-specific logic.

INTERVIEW TRAP. "But it's just code — code isn't data." That is the wrong instinct, and Q7 exists entirely to answer it.

Q7. Why is "it's just code, not data" wrong?

SHORT ANSWER. Because the code is the confidential asset in our world. A derivation program encodes the endpoint definition, the analysis population rules, the imputation strategy, the censoring rules — that is the SAP made executable. And in an unblinded study, the code can carry the unblinding itself: a treatment-mapping format, a hardcoded arm label, a condition on a group variable.

FOR THE SAS PROGRAMMER. You have written a program with `if trt01pn = 1 then ...` and a comment naming the arm. You have written a format that maps a randomisation code. If that program is confidential in your document management system, it is confidential in a prompt box. Nothing about pasting it changes its classification.

DETAIL. Three specific ways code leaks something that matters:

1. **Derivation logic is IP.** A novel composite endpoint derivation, an adaptive design's interim rules, an estimand implementation — these are competitively sensitive and often the sponsor's most protected material in the whole analysis package.

2. **Code can be unblinding.** Not hypothetically: hardcoded treatment labels, group-conditional logic, and mapping formats appear in real programs. A programmer who is blinded pasting code that carries an arm mapping has just done something worse than sharing a dataset.
3. **Comments and headers.** Study number, compound code, indication, sponsor name, milestone dates. A header block is a competitive-intelligence summary. "Compound XYZ-123 Phase 3 database lock 12-Sep" tells an observer a great deal.

Also worth saying: even sanitised, the pattern of questions is informative. A burst of prompts about non-proportional-hazards methods and RMST from one sponsor's account in one week is a signal. That is a reason to prefer a private tenancy over anonymous individual accounts, not a reason to ban the tool.

INTERVIEW TRAP. "Would you allow pasting a SAS program to be translated to R?" My answer: yes if it's a generic utility with no study logic; for a study derivation, only inside the approved private tenancy under zero-retention terms; never from a personal account. The trap is a candidate who has only thought about data and never realised the code is the crown jewels.

Q8. What deployment models make this acceptable, and what contract terms would you require?

SHORT ANSWER. An enterprise tenancy under commercial terms with no training on your inputs, contractual zero data retention or an explicitly bounded retention window, SSO with your identity provider, and an audit log you own. Self-hosted or cloud-vendor-hosted models inside your own subscription are the strongest option; consumer accounts are never acceptable for anything study-related.

FOR THE SAS PROGRAMMER. This is the same diligence you ran the first time a CRO wanted to hold your data in their environment, or the first time analysis moved to a hosted platform. You asked where the data sits, who can see it, how long it's kept, and what happens at exit. Same four questions.

DETAIL. Verified as of 2026-08-02, and stated with the caveat that these terms move fast and must be confirmed with the vendor's account team rather than from a blog:

- **Training.** Major providers do not train on commercial/API customer inputs by default. Anthropic's commercial terms (Claude for Work / Enterprise / API) exclude commercial customer data from model training, with no user-level toggle to get it wrong. The OpenAI API does not train on customer data by default. The consumer products are a different agreement — this is the distinction people miss.
- **Retention.** Default API retention is commonly around 30 days for abuse monitoring. Zero data retention is a negotiated configuration, not a checkbox everywhere, and it has real edge cases: some Anthropic "covered models" require 30-day retention and cannot be used from a ZDR-enabled organisation at all; OpenAI's ZDR is configurable at organisation and project level with carve-outs for certain classifier hits.
- **HIPAA/BAA.** Available from both, but narrower than people assume, and scoped to particular products and configurations — Anthropic has decoupled its HIPAA-ready path from ZDR, and coverage excludes several surfaces. If you buy through a cloud marketplace, the cloud provider is your processor and the model vendor's BAA may not apply.

- **Deployment.** Model-in-your-cloud options (Bedrock, Vertex/Google Cloud, Azure) keep inference inside your subscription and your existing cloud DPA, which is often an easier internal sell than a new vendor relationship.

The honest caveat to state in the room: zero-retention is a contractual and architectural assurance you generally cannot independently audit. You are accepting a supplier assurance, exactly as you do for every SaaS system you use. Say that out loud — it is the mark of someone who has actually been through a vendor assessment.

INTERVIEW TRAP. "Isn't on-premise the only safe answer?" No, and answering yes marks you as someone who has never costed it. Self-hosting a capable model means owning the GPUs, the model updates, the security patching and the evaluation — a real quality burden traded for a real quality burden. A private tenancy with contractual ZDR under an enterprise agreement is where most large pharma has actually landed.

Q9. A programmer on your team pasted an ADaM spec into a public chatbot. What do you do?

SHORT ANSWER. Treat it as a confidentiality incident, not a disciplinary one — first. Contain, assess, report through the quality system, and fix the reason it happened. If people are afraid to tell you, you lose the ability to contain the next one, and the next one will be data instead of a spec.

FOR THE SAS PROGRAMMER. Identical playbook to an unblinding near-miss or an emailed dataset to a wrong address. You have run this. Scope it, notify, document, CAPA.

DETAIL. Sequence: (1) establish exactly what was pasted and from which account; (2) request deletion via the vendor's data-deletion path and record the request; (3) notify information security and the study's confidentiality owner — for a CRO, this is very likely a contractual notification to the sponsor; (4) log a deviation; (5) root cause. The root cause is nearly never "bad programmer." It is that the approved tool was slow, hard to access, or not available for their task, so they used the fast one. The CAPA that works is making the compliant path the easy path, plus one targeted training update.

Say the last part explicitly, because it is the governance-level answer: a ban with no sanctioned alternative produces shadow usage, which produces incidents you cannot see. A sanctioned, convenient, monitored tool produces incidents you can see. Given a hostile choice between the two, the second is safer, and that is a defensible argument to make to a QA lead.

INTERVIEW TRAP. "Would you report it to the sponsor?" For a CRO, almost certainly yes — check the CDA, but the instinct that protects you is disclose early. Candidates who hesitate here are telling the interviewer how they would behave under pressure.

Section A3 — Reproducibility, audit trail and the limits of the tool

Q10. If a model wrote the code, can you reproduce it?

SHORT ANSWER. You cannot reproduce the generation, and you don't need to. You reproduce the code, which is a file in version control, deterministic, and reruns identically forever. The thing that is not reproducible is the model's output for a given prompt — and that is exactly why the code has to be committed, reviewed and owned rather than regenerated on demand.

FOR THE SAS PROGRAMMER. Nobody ever asked you to reproduce your own thought process while writing a program. They asked you to produce the program, the log, the output, and the QC evidence. Same here. The model sits where your thinking sat.

DETAIL. Large language models are not deterministic in practice even at temperature zero — floating-point non-associativity in batched GPU inference, hardware and kernel variation, and silent model-version updates all mean the same prompt can produce different code on different days. Two consequences that matter and that almost no candidate articulates:

1. **Never put a model call inside a production pipeline** that generates an analysis artefact. If step 4 of your ADaM build asks a model to do something, your pipeline is no longer reproducible and you have created an in-scope AI system with a credibility burden. Generate code offline, commit it, run the committed code.
2. **Do not rely on regenerating.** "We can always ask it again" is not a recovery plan. The artefact of record is the committed file with a human author, a reviewer, and a tag.

This also disposes of a bad idea that circulates: storing prompts as the source of truth so you can "rebuild the code later." Prompts are useful context; they are not source. The source is the source.

```
# Reproducibility boundary — state this crisply
#
#   NOT reproducible          |  REPRODUCIBLE (and what you archive)
#   ------------------------- | ---------------------------------------
#   the model's output for a  |  the committed .R file
#   given prompt              |  the renv.lock / package snapshot
#   the model version behind  |  the container or R version
#   an API endpoint           |  the input data (frozen extract)
#                             |  the run log and the tagged commit
#
# Rule: no LLM call inside a pipeline that produces a regulatory artefact.
```

INTERVIEW TRAP. "So AI breaks reproducibility?" No — AI in the pipeline breaks reproducibility; AI in the authoring does not, because the output of authoring is a static file. Drawing that line cleanly is the whole answer, and it is the answer a statistician in the room will respect.

Q11. What's your audit trail when AI assistance is used?

SHORT ANSWER. The same audit trail as always — Git history, pull-request review record, QC documentation — plus one addition: a flag in the commit or PR recording that AI assistance was used. The human is the author of record in every system. I would not create a separate AI-code register; it fragments the evidence and adds a control nobody maintains.

FOR THE SAS PROGRAMMER. Your audit trail has always been the change record, the reviewer, and the QC evidence. Nothing is removed. One field is added, and only so you can answer a question honestly if an inspector or a sponsor asks it.

DETAIL. Concretely, a `Assisted-by:` trailer in the commit message or a checkbox in the PR template is enough. It costs nothing, it is queryable, and it means that if the landscape changes — if a regulator or a sponsor starts asking — you can answer with data instead of a survey. That is a cheap option on an uncertain future, which is the right way to buy governance.

What you do not do is treat it as a separate approval gate, because that creates an incentive not to tick the box. Make it observational and non-punitive or the data is worthless.

```
# Commit trailer — cheap, queryable, non-punitive
feat(adsl): derive treatment-emergent flag per SAP v3.1

Implements TRTEMFL per SAP section 9.2. Independent QC by JS,
compare clean against QC dataset, run 2026-07-14.

Assisted-by: AI coding assistant (enterprise tenancy)
Reviewed-by: J. Smith <j.smith@example.com>
```

INTERVIEW TRAP. "Doesn't 21 CFR Part 11 require you to capture that?" Part 11 governs electronic records and signatures for records the predicate rules require you to keep. Whether analysis source code is such a record is genuinely debated — many organisations govern code under GCP and their QMS rather than as a Part 11 record, and **the industry has not settled this**. Do not assert Part 11 applies to your Git history; say that your Git history is designed to satisfy it either way, which is true and unfalsifiable in a good way.

Q12. Where does AI genuinely help in clinical programming?

SHORT ANSWER. Boilerplate, translation, tests, documentation, and comprehension. Those five. It is very good at things where I can check the answer quickly and where being 90% right saves real time, and much worse at things where being 90% right is dangerous.

FOR THE SAS PROGRAMMER. Think about which parts of your week are typing and which are judgement. It compresses the typing. It does not do the judgement, and every disappointment people have with it comes from asking it to.

DETAIL. The concrete high-value uses:

- **Boilerplate and scaffolding.** Function skeletons, roxygen headers, `tryCatch` wrappers, a `targets` pipeline stub, a Shiny module shell, repetitive `dplyr` chains across 30 similar TLFs.
- **SAS-to-R translation.** This is the killer application for a team like yours. It handles `proc sort` + `data` step merges, `retain` / `by`-group logic, and array processing into `dplyr` / `tidyr` idiom competently. Treat the output as a first draft to be verified against the original's results, not read for equivalence.
- **Test cases.** Genuinely underrated. Ask it to enumerate edge cases for a derivation — partial dates, records outside the treatment window, subjects with no post-baseline value, ties in a `last` observation — and it produces a list a tired human would not. You then implement the ones that matter.
- **Documentation.** Roxygen comments, README files, migration notes, the narrative half of a validation document. You still own the content.

- **Explaining unfamiliar code.** Someone left; there is a 900-line legacy program and no spec. "Explain what this does and list its assumptions" is a real hour saved, and the answer is verifiable by reading the code.
- **Review assistance.** As a second pair of eyes on a diff, not the first. Good at spotting a missing `na.rm`, an unhandled join explosion, an off-by-one in a window.

The pattern across all six: cheap to verify, expensive to type.

INTERVIEW TRAP. Naming only "it writes code faster" makes you sound like you read a press release. Naming test-case generation and legacy-code comprehension makes you sound like you have used it on a real deliverable, because those are the uses people discover in month three.

Q13. Where is it dangerous?

SHORT ANSWER. Anywhere the output is plausible and the error is silent. Statistical method selection, derivation logic that looks right, hallucinated function arguments, and anything where I would have to do as much work to verify it as to write it. My rule: if I cannot check it faster than I can build it, I don't use it for that.

FOR THE SAS PROGRAMMER. The dangerous CRO deliverable was never the one that failed loudly. It was the one that ran clean, produced a table, and was subtly wrong in the population definition. That is exactly the failure mode here, at higher volume.

DETAIL. The specific hazards, in the order they will bite a clinical team:

- **Silently plausible derivations.** It will confidently produce a baseline-flag derivation that uses the wrong tie-breaker, or a treatment-emergent window that is off by a day, or a last-observation rule that picks the wrong record when dates tie. It reads beautifully. The only defence is comparing against the specification and against independent programming — which you were doing anyway.
- **Hallucinated arguments and functions.** Confidently calling `dplyr::filter(.by = )` in a version that doesn't have it, or inventing an argument to a `pharmaverse` function. Cheap to catch, because it errors — this is the safe kind of wrong.
- **Statistical methodology.** Never let it choose a method. Ask it to implement a method the statistician chose, and ask it to explain a method you're learning, but a model's opinion on whether to use MMRM or a mixed model with a different covariance structure, or how to handle an estimand's intercurrent events, is not an input to a SAP. This is where a hallucination has consequences no QC step catches, because QC checks that you implemented what you said, not whether what you said was right.
- **Regulatory and CDISC specifics.** It will state CDISC controlled terminology or IG rules with total confidence and get versions wrong. Check against the actual IG.
- **Anchoring.** The subtlest one. Once you have read a plausible draft, your review is contaminated — you are now checking that draft rather than thinking independently. For genuinely critical derivations, write your version first, then compare.

INTERVIEW TRAP. "How do you stop your team from over-trusting it?" The good answer is structural, not exhortatory: independent programming stays independent — the QC programmer must not see the production code, and ideally does not use the same assistant conversation. If both sides of a double-programming check come from the same model, you have not double-programmed anything, you have run the same generator twice. That single point is probably the highest-value thing in this entire module.

Section A4 — Governance, and handling the room

Q14. Present the risk register you'd take to a QA lead.

SHORT ANSWER. Six risks, honestly rated, each with a control I can evidence. I'd rather walk in with the risks named than have QA discover them, because the credibility I need is for the next proposal as much as this one.

FOR THE SAS PROGRAMMER. Same document you have written for any new process or system. Risk, likelihood, impact, control, residual, owner.

DETAIL. The register I would actually present:

RISK	CONTROL	RESIDUAL
1. Confidential data or code leaves the sponsor boundary	Enterprise tenancy, SSO, no-training and zero/bounded retention terms; DLP on the egress path; acceptable-use training	Low
2. Plausible-but-wrong derivation reaches a deliverable	Unchanged double programming; QC blinded to production code; spec-based review	Low-Med
3. Reviewer complacency — the draft reads too well (the risk nobody writes down)	AI-assist flag is observational not a gate; periodic review-quality sampling; reviewer training on failure modes	MEDIUM
4. Double programming collapses to a single generator	Policy: QC programmer works from the spec independently; separate sessions/tools	Low
5. Shadow usage on personal accounts if we ban or throttle	Provide a fast sanctioned tool; monitor; non-punitive incident reporting	MEDIUM
6. Skills atrophy in junior staff	Juniors write first, then compare; AI not used as the teaching path for core skills	Medium

Note which two are rated highest. They are human-factors risks, not technical ones, and saying that to a QA lead is what gets the proposal taken seriously. The technical controls are easy; the behavioural ones are the actual programme.

INTERVIEW TRAP. "What's the residual risk you'd accept?" Name one honestly. Mine: I accept that I cannot independently verify the vendor's retention assurance, on the same basis I accept it for every other SaaS system in the validated estate. A candidate who claims zero residual risk has not run a risk assessment, they have written a marketing document.

Q15. The interviewer is clearly hostile to AI. What do you say?

SHORT ANSWER. Agree with the substance of their objection before you say anything else — because they are usually right about the specific risk they are naming, and they are testing whether you have judgement or enthusiasm. Then reframe: my job is not to advocate for a tool, it is to make sure that when the organisation adopts it, the controls exist first.

FOR THE SAS PROGRAMMER. You have sat opposite a QA director who was hostile to a change you needed. You did not win by out-enthusing them. You won by showing you had thought about the failure modes more carefully than they had.

DETAIL. A workable sequence:

1. **Concede the real point.** "You're right that these tools produce confident, wrong answers, and in our domain a confident wrong answer is the worst kind."
2. **State the accountability position.** "Which is why nothing changes about who's accountable, or about double programming, or about review."
3. **Give the boundary you would personally hold.** "I would not put a model inside a pipeline that generates a submission artefact. Ever. That's a hard line for me."
4. **Reframe the risk direction.** "The risk I'd actually raise with you isn't people using it — it's people using it on personal accounts because we haven't given them a sanctioned path. That's the version I can't see or control."
5. **Leave the door open.** "If the answer here is 'not yet', that's a defensible position and I'd implement it properly."

Never argue that they are behind the times. And notice that this answer is fully compatible with three of five live senior reqs demanding AI-enabled tooling — you are demonstrating exactly the judgement those reqs actually want, which is not enthusiasm.

INTERVIEW TRAP. The hostility is often a test rather than a position, and sometimes the hostile person is the QA lead you'd need as an ally. Candidates who capitulate entirely ("I'd never use it") fail the JD; candidates who push back hard fail the culture. Concede, then hold one clear line — that combination reads as senior.

Q16. The interviewer is an enthusiast and wants you to push the frontier. What do you say?

SHORT ANSWER. Match the energy on the upside and then be the person in the room who names the boundary — because at Principal/AD level they are hiring judgement, and an enthusiast who has never said no to anything is not who you want owning validation governance.

FOR THE SAS PROGRAMMER. Same as when a sponsor wants a shortcut you can deliver but shouldn't. You say yes to the goal and no to the specific unsafe route, and you offer the safe route in the same breath.

DETAIL. What to be genuinely enthusiastic about, because it is genuinely good: the SAS-to-R migration acceleration, which is the single biggest cost line in every migration programme; automated first-draft test generation across a standards library; documentation debt cleared at a rate no team ever funds manually; onboarding time onto an unfamiliar legacy codebase collapsing.

Then the boundary, delivered without hedging: "The line I hold is that no model call sits inside a pipeline producing a regulatory artefact, and independent QC has to stay genuinely independent of the generator. Everything else I'll pilot and measure."

If they ask what you'd pilot first, have an answer ready: SAS-to-R translation of a standards library, because it has a natural oracle — the SAS output already exists, so you can measure success objectively rather than by feel. That answer signals that you think in terms of measurable pilots, which is exactly the Agile-delivery competence the GSK/ViiV req is asking about.

INTERVIEW TRAP. The enthusiast may float something genuinely unsafe — an agent that generates and runs TLFs end to end, or a model that adjudicates QC discrepancies. Say no, and say why in one sentence: no deterministic reproduction path, and no independent check. Being able to disagree with an enthusiastic hiring manager politely and concretely is worth more than the rest of the answer.

Q17. What does good prompt discipline look like as a professional skill?

SHORT ANSWER. Give it the specification, constrain the environment, ask for the smallest verifiable unit, and never accept output you can't test. The discipline is mostly the same discipline as writing a good ticket for a junior programmer — and the people who are good at one are good at the other.

FOR THE SAS PROGRAMMER. You already know that a vague request to a junior produces a vague program. You learned to write "derive TRTEMFL per SAP section 9.2, using ADSL.TRTSDT, edge case for partial start dates is X." That instruction is a prompt.

DETAIL. What separates competent from careless use:

- **Give it the spec, not the vibe.** Paste the derivation rule from the approved specification (from the approved tenancy, per Q6) rather than describing it from memory. Most bad output is bad input.
- **Constrain the environment.** State the R version, the allowed packages, and the house style. Otherwise it reaches for whatever was most common in its training data, which may be a package that isn't in your validated snapshot.
- **Small units.** One function, one derivation. A whole ADaM domain in one prompt produces something impressive-looking that nobody can review properly.
- **Ask for the assumptions.** "List the assumptions you made and the edge cases you did not handle" surfaces more real defects than reading the code does, and it takes ten seconds.
- **Never accept what you can't test.** If you can't state how you would know it's wrong, don't merge it.
- **Watch the anchoring.** For critical logic, write yours first.

INTERVIEW TRAP. "Isn't prompt engineering a fad?" Partly — the specific tricks age out every few months. What doesn't age is precise specification of intent and rigorous verification of output, which is a description of professional programming and of clinical QC. Say that. It reframes the skill as something an 11-year clinical programmer already has, which is both true and exactly what the interviewer wants to hear.

Part B — Git, reproducibility and standardization at portfolio scale

Section B1 — Git for someone who has only ever used a shared drive

Q18. Explain Git to someone whose entire career has been a network drive and file-naming conventions.

SHORT ANSWER. Git is a system that records every version of every file, who changed it, when, and why — automatically, with no discipline required from the person saving. Instead of `adsl_v3_final_JS_REVISED.sas`, there is one file called `adsl.R` and a complete, immutable history behind it that you can inspect or return to at any point.

FOR THE SAS PROGRAMMER. Your naming convention is a version control system. It's just one implemented in human memory, enforced by nagging, with no audit trail and no way to answer "what exactly changed between v2 and v3." Git is the same intent, done by the machine, with the answers built in.

DETAIL. Six words, and you only need six:

- **Repository (repo)** — a folder whose entire history is tracked. Roughly "the study's programming area."
- **Commit** — a saved snapshot of the whole repo at a moment, with an author, a timestamp, and a message. This is the unit of change and the unit of audit. It is closest to a signed, dated change record.
- **Branch** — a parallel line of work. You branch to make a change without disturbing the version everyone else is using. There is nothing to copy and nothing to merge back manually.
- **Merge** — folding a branch's changes back into the main line.
- **Pull request (PR) / merge request** — a proposed merge, opened for review. Someone must read it and approve before it lands. This is the control point, and it is where double programming meets modern tooling.
- **Remote** — the shared server copy (GitHub, GitLab, Bitbucket, Azure DevOps). Your laptop has a full copy; the remote is the authoritative one.

The mental shift that takes longest: the history is immutable and complete, so you stop being careful about saving and start being careful about committing well. You never again need a folder called `archive`.

```
git clone https://git.example.com/studies/ABC-301.git   # get the study repo
git checkout -b feature/adsl-trtemfl                    # start a change
# ... edit adsl.R ...
git add adam/adsl.R
git commit -m "derive TRTEMFL per SAP v3.1 section 9.2"
git push -u origin feature/adsl-trtemfl                 # send it to the server
# then open a pull request in the web UI for review
```

INTERVIEW TRAP. Do not oversell Git as an audit trail that satisfies regulators by itself. Git history is trivially rewritable by design (`rebase`, `push --force`, amended commits). What makes it a control is the server-side configuration: protected branches, force-push disabled, required reviews, identity tied to SSO. Saying that unprompted is the difference between someone who has used Git and someone who has governed it.

Q19. What goes in version control, and what must never?

SHORT ANSWER. Code, specifications, configuration, documentation, and environment manifests go in. Data never goes in — not raw, not derived, not "just a small test extract." Credentials never go in. Outputs generally do not go in, because they are regenerable and they bloat the history.

FOR THE SAS PROGRAMMER. Same separation you already keep between the programs area and the data area, with the same reasoning: different access controls, different retention, different size profile, different regulatory treatment.

DETAIL. Why data is a hard no, in the order that matters:

1. **You cannot take it back.** Git history is designed to be permanent. A patient-level dataset committed once is in every clone, on every laptop, forever. Removing it means rewriting history across every copy, which is an incident with a project plan attached.
2. **Access control mismatch.** Repo access is usually broad within a team. Patient-level and unblinded data access is deliberately narrow.
3. **Size.** Git handles text diffs; it stores binary datasets as whole new copies on every change.

Instead, reference data by path and by a frozen extract identifier, and record the extract's checksum in the repo. That gives you traceability without custody.

```
# .gitignore for a clinical study repo
*.sas7bdat
*.xpt
*.rds
*.parquet
data/
output/
*.log
*.lst
.Renviron          # credentials live here, and here is never committed
.Rhistory
.Rproj.user/
renv/library/      # the library itself is rebuilt from renv.lock, not stored
```

Credentials belong in the environment or a secrets manager, never in a file that gets committed. If one is committed, treat it as compromised and rotate it — removing it from history is not sufficient, because it was already cloned.

INTERVIEW TRAP. "What about the final outputs — shouldn't they be versioned for the submission?" Outputs belong in the controlled document/output system with its own retention, not in Git. What Git gives you is the ability to regenerate them from a tagged commit plus a frozen extract plus a pinned environment. Confusing "versioned" with "archived" is a common and revealing mistake.

Q20. How would you structure repositories across a portfolio — one per study, or a monorepo?

SHORT ANSWER. Per-study repositories for study code, plus a small number of separate repositories for shared standards packages. That combination gives you clean per-study access control, a clean lock/submission tag per study, and a single versioned source for anything reused across studies.

FOR THE SAS PROGRAMMER. It mirrors what you already have: a study area per study, and a global macro library maintained centrally with its own versioning and release process. You have never wanted one folder containing forty studies, and the reasons haven't changed.

DETAIL. The argument for per-study repos in this domain is mostly regulatory rather than technical:

- **Access control follows the study.** Blinded/unblinded segregation, sponsor segregation at a CRO, and study team membership are all naturally per-study.
- **Tagging is meaningful.** A tag `dbl-2026-05-14` means something if the repo is one study. In a monorepo it means nothing, because a single tag spans forty studies at forty different states.
- **Archival and retention follow the study.** A study closes; its repo is frozen and retained. You cannot freeze one-fortieth of a monorepo.
- **Blast radius.** A mistake in one study repo cannot touch another.

Shared code goes the other way — one repo per standards package, versioned and released like software, consumed by studies as a pinned dependency. At compound or programme level, a compound-level repo for shared derivations across studies of the same molecule is a sensible middle layer.

```
org/
  std-adam/            # internal standards package: ADaM derivation helpers
  std-tlf/             # internal standards package: table/figure templates
  std-templates/       # study repo skeleton, used to start every new study
  compound-XYZ/        # shared across the XYZ programme
  ABC-301/             # study repo   -- consumes std-adam v2.4.0 (pinned)
  ABC-302/             # study repo   -- consumes std-adam v2.3.1 (pinned)
```

Note that ABC-301 and ABC-302 are pinned to different versions of the standards package. That is correct and deliberate — see Q31.

INTERVIEW TRAP. "Doesn't a monorepo make standardization easier?" It makes propagation easier and control harder, and in a validated environment control wins. The follow-up worth pre-empting: "so how do you stop forty studies drifting?" You don't stop drift, you make it visible — a dashboard of which study is pinned to which standards version is the governance artefact, not a rule that everyone must be current.

Q21. Which branching strategy fits clinical delivery?

SHORT ANSWER. Short-lived feature branches off `main`, merged through a reviewed pull request, with tags marking milestones — essentially GitHub Flow rather than full GitFlow. Clinical programming has no continuous release train and no hotfix-to-production-while-developing-v2 problem, so GitFlow's `develop` / `release` / `hotfix` machinery is ceremony you'll pay for and not use.

FOR THE SAS PROGRAMMER. A feature branch is the equivalent of taking a copy of a program to work on a change while the current version stays usable by the team. The difference is that merging back is mechanical and reviewed rather than manual and hopeful.

DETAIL. What actually fits:

- **`main` is always the current best version** of the study's code. Protected: no direct pushes, no force-push, at least one approving review required.

- **Branch per change**, named for the work — `feature/adsl-trtemfl`, `fix/ae-window-boundary`. Short-lived, days not weeks. Long-lived branches produce painful merges, and painful merges produce mistakes in exactly the code you most need to be right.
- **Tag at milestones, don't branch at them.** A tag is an immutable pointer to a commit — perfect for "this is the code as of database lock."
- **Branch after lock only if you must** re-run for a post-lock question while forward work continues. That's the one place a release branch earns its keep.

```
git tag -a dbl-2026-05-14 -m "Code as of database lock, protocol ABC-301"
git tag -a submission-nda-2026-09 -m "Code frozen for NDA submission"
git push --tags

# two years later, a regulator asks a question:
git checkout dbl-2026-05-14        # exact code at lock
renv::restore()                    # exact package versions at lock
```

Worth knowing that this is live industry work, not settled doctrine: PHUSE runs a working group, The Use of Git in Statistical Programming (Emerging Trends and Innovation), whose deliverables include principles, example workflows in the context of the current QC process, and a white paper covering the regulator's point of view and a proposed alternative QC process built on Git features. As of 2026-08-02 the white paper is still forthcoming. Git adoption in statistical programming lags other industries badly — the working group's own framing is that Git is above 90% adoption among programmers generally and the pharmaceutical industry is a notable laggard. **The industry has not settled what QC looks like in a Git-native world**, and saying that, with the working group as evidence, is a much better answer than pretending there is a standard.

INTERVIEW TRAP. "Why not trunk-based with direct commits?" Because the reviewed pull request is your control point, and losing it loses the mapping to double programming. Pure trunk-based development depends on a strong automated test suite standing in for human review — most clinical teams do not have one yet. Say "trunk-based is where I'd want to get to once the automated checks are real; today the PR is the control."

Section B2 — Review as a control, and mapping to the SAS world

Q22. What is a reviewer actually checking in a pull request?

SHORT ANSWER. Four things, in this order: does it do what the specification says; is it correct at the edges; is it maintainable by someone else; and is anything in it that shouldn't be committed. Style is last and should mostly be automated away.

FOR THE SAS PROGRAMMER. This is your code review step, with a better interface. The one genuine addition is that the reviewer sees a diff — precisely what changed since the last approved state — rather than re-reading a whole program and trying to remember what was different.

DETAIL. A PR template turns this from a personal habit into a control you can evidence:

```
## What changed and why
Implements TRTEMFL for ADAE per SAP v3.1, section 9.2.

## Specification reference
ADaM spec ABC-301 v1.4, ADAE, row 34.

## How it was verified
[ ] Independently programmed by ..... ; results compared, no differences
[ ] Edge cases tested: partial start dates, subjects with no post-baseline,
    events on the day of first dose, events after last dose + 30d
[ ] Full study rerun clean, log reviewed, no WARNING or ERROR

## Checks
[ ] No data files, no credentials, no patient-level content committed
[ ] renv.lock unchanged (or: change justified below)
[ ] Assisted-by trailer set if AI assistance was used
```

That template is a controlled record. It is timestamped, attributable, immutable once merged, and it links the change to the specification and to the QC evidence. Show an inspector the PR history for a study and you have shown them the change control for the analysis in a form they can actually navigate — which is more than most SAS-era shared drives could offer.

INTERVIEW TRAP. "Is a PR approval the same as double programming?" No, and be firm. Review is a read of one implementation. Double programming is a second independent implementation compared on results. They catch different defects — review catches unmaintainable and obviously wrong code; double programming catches a correct-looking implementation of the wrong rule. For a critical derivation you need both, and a team that has quietly replaced double programming with PR review has downgraded its control while feeling more modern.

Q23. How does the SAS-world double programming model map onto a Git workflow?

SHORT ANSWER. Cleanly, with one design decision to get right: the QC programmer must work from the specification, not from the production code. In Git terms, they implement in their own branch without reading the production branch, and the comparison of results is the merge gate.

FOR THE SAS PROGRAMMER. Structurally identical to what you do now. What changes is that the independence is enforceable and evidenced — you can prove from the history who wrote what, when, and in what order, rather than asserting it.

DETAIL. A workable pattern:

- Production programmer works on `feature/adae-trtemfl`.
- QC programmer works on `qc/adae-trtemfl`, from the spec, and their code and results live in the repo too — QC code is a deliverable, not a scratch file.
- The comparison (in R, typically `diffdf` or a `compare`-style check) is run and its output attached to the PR.
- The PR merges only with the comparison clean and the reviewer's approval.

Two subtleties that get missed. First, if your platform shows the QC programmer the production diff by default, independence is compromised by the tooling — configure it, or keep QC work in a separate repo area until comparison. Second, and increasingly important: if both programmers use the same AI assistant in the same way, independence is compromised regardless of process (see Q13). That is a genuinely new consideration and worth raising unprompted.

INTERVIEW TRAP. "Isn't this heavier than the SAS process?" It is the same weight; the difference is that the evidence is generated automatically rather than assembled at audit time. If the interviewer is worried about cost, the honest answer is that the cost is in the migration and the training, not in the steady state.

Q24. How do you prevent someone rewriting history to hide a change?

SHORT ANSWER. Server-side controls, not trust. Protected branches, force-push disabled on `main` and on tags, signed or SSO-attributed commits, required reviews, and administrators who cannot bypass those rules without leaving an entry in the platform audit log.

FOR THE SAS PROGRAMMER. This is the same question as "who can edit the validated program area outside change control." The answer is the same: nobody, and the system enforces it rather than the SOP asking nicely.

DETAIL. The configuration that makes Git a control rather than a convenience:

```
Branch protection on `main`:
  - require pull request before merging
  - require at least 1 (or 2, for critical studies) approving review
  - dismiss stale approvals when new commits are pushed
  - block force pushes
  - block branch deletion
  - require status checks to pass (lint, unit tests, renv consistency)
  - include administrators (no bypass)
Tags:
  - protect release/lock tags against deletion and re-pointing
Identity:
  - SSO-enforced; commits attributable to a named, authenticated user
Retention:
  - platform audit log exported and retained per record-retention SOP
```

The important point to make: an individual's local Git history is freely rewritable, and that's fine and intentional — the local branch is a working area. The server's protected branch is the record.

Explaining that distinction is the single clearest signal that you understand Git as governance rather than as a tool you were told to use.

INTERVIEW TRAP. "So is Git 21 CFR Part 11 compliant?" Git is not compliant or non-compliant; a system is. A Git platform configured with SSO-attributed identity, immutable protected branches, retained audit logs and enforced review can support Part 11-style expectations, and industry papers going back to the PHUSE 2017 paper Git with It and Version Control! have made that argument. But compliance is a property of your configured, qualified system plus your SOPs, not of the software. Never say "Git is Part 11 compliant" — it is the kind of sentence that ends your credibility with a QA audience.

Q25. Your team has never used Git. How do you introduce it without stopping delivery?

SHORT ANSWER. Start with one non-critical study or the standards library, not the portfolio. Teach five commands, not the manual. Put the platform behind SSO from day one so you never have to retrofit governance. And expect three months of slower delivery on the pilot, and say so up front rather than discovering it.

FOR THE SAS PROGRAMMER. You have run process rollouts. The failure mode is always the same: mandating a tool before anyone has a working pattern to copy, then dealing with forty different interpretations.

DETAIL. The sequence that works:

1. **Pilot on the standards library**, not a live study. It's low-risk, it's the code that most benefits from versioning, and it produces the internal expertise you'll need.
2. **Five commands only** — `clone`, `checkout -b`, `add` / `commit`, `push`, and open a PR in the web UI. Everything else on demand. Most clinical programmers never need `rebase`, and teaching it early causes accidents.
3. **A template repo** so nobody designs their own layout. This is the highest-leverage single artefact in the whole rollout.
4. **A named owner** who unblocks people the same day. Merge conflicts are frightening the first time and trivial the fifth.
5. **Governance from day one** — SSO, protected `main`, PR template. Retrofitting controls onto an established informal practice is much harder than starting with them.

Be honest about the cost: the first study will be slower. The gain arrives at the second and third, and compounds at the portfolio level where a fix to a standard propagates by version bump rather than by forty manual copies.

INTERVIEW TRAP. "What's the hardest part?" Not the tool. It is that Git exposes work in progress to colleagues, and people who have spent a career polishing a program before anyone sees it find that genuinely uncomfortable. Naming a human obstacle rather than a technical one is what a hiring manager at this level is listening for.

Section B3 — Reproducibility and environments

Q26. What does reproducibility actually require?

SHORT ANSWER. Same code, same data, same environment, same result. Three legs. Everyone remembers code and most people remember data. The leg that gets forgotten is the environment — the R version and the exact versions of every package — and it is the one that breaks first, because packages update underneath you whether or not you touched anything.

FOR THE SAS PROGRAMMER. You have partly been protected from this by SAS's stability. A SAS 9.4 program written in 2016 generally still runs in 2026. R's ecosystem moves much faster and has no equivalent guarantee, so what SAS gave you by default, R makes you configure explicitly. That is the single biggest cultural adjustment in the migration, and it is worth saying so — it shows you understand why R teams talk about environments constantly and SAS teams never did.

DETAIL. The three legs, and what each one costs you when it's missing:

1. **Code.** Solved by Git plus a tag. Cheap.
2. **Data.** A frozen, identified extract with a checksum, referenced from the repo. Costs you discipline about extract management, which most clinical organisations already have.
3. **Environment.** R version, package versions, and where relevant the OS and system libraries. Solved by `renv` plus a pinned repository snapshot, and at the strongest level by a container image.

The failure mode is specific and worth describing because it makes the abstraction concrete: a `dplyr` or `tidyr` version bump changes how a grouped operation handles an edge case, or a new default in a modelling package alters a contrast. Your code is unchanged, your data is unchanged, and your table has moved. Nothing in your QC catches it, because nothing changed in anything you version-controlled. This is not hypothetical — the R Submissions Working Group's FDA pilots surfaced real environment-replication problems on the reviewer side, resolved with `renv` and `pkglite`.

```
Reproducible run =
    tag         : dbl-2026-05-14        (code, from Git)
  + extract     : ABC301_RAW_2026-05-14 (data, frozen + checksummed)
  + renv.lock   : R 4.3.2, 187 packages at pinned versions
  + repository  : PPM snapshot 2026-01-15  (where those versions come from)
  [+ image      : ghcr.io/org/abc301:1.2.0 — the strongest form]
```

INTERVIEW TRAP. "Which leg do people forget?" Say environment, immediately, and then say why: because it degrades silently and with no change on your side. That single observation is the whole answer, and it's the one that lands.

Q27. Explain `renv`, and how you'd restore a study environment two years later.

SHORT ANSWER. `renv` gives a project its own private package library and records every package and version in a lockfile, `renv.lock`, which is committed to Git. Two years later you check out the tag, run `renv::restore()`, and it rebuilds the exact library. That is the mechanism that makes "same environment" a file rather than a hope.

FOR THE SAS PROGRAMMER. The nearest analogue is pinning the SAS version and the exact version of the global macro library that a study was run against — except `renv` does it for all several hundred dependencies automatically, including the ones you never chose and never heard of.

DETAIL. The workflow is three commands and one discipline:

```
renv::init()      # create a project library; snapshot what's currently used
renv::snapshot()  # after adding/updating a package, re-record renv.lock
renv::restore()   # rebuild the library exactly as renv.lock specifies

# two years later, answering a regulatory question:
#   git checkout dbl-2026-05-14
#   renv::restore()
```

The discipline: `renv.lock` is committed, and a change to it is a reviewed change like any other. A PR that silently bumps forty package versions alongside a one-line derivation fix is a red flag, and your PR template should make it visible.

What `renv` does **not** cover, and you should say this unprompted because it separates people who have read the docs from people who have run it in anger:

- The **R version itself.** `renv.lock` records it but cannot install it. You need `rig`, a module system, or a container.
- **System libraries.** A package that links against a system OpenSSL, a Java runtime, or a specific `libxml2` will fail to build on a machine that has moved on.
- **Source availability.** `renv::restore()` fetches from a repository. If a package version has been archived off CRAN and your repository has no snapshot of it, restore fails. This is the real-world reason to use a snapshot-capable internal repository rather than pointing at live CRAN. Practitioners are also candid that `renv` "can sometimes fail catastrophically" on complex restores, which is one reason teams with a managed platform often prefer platform-level snapshots over per-project `renv`.

The robust answer for a regulatory-question-in-two-years scenario is therefore: `renv.lock` for the record, a pinned internal repository snapshot so the sources still exist, and a container image for anything where the OS layer matters.

INTERVIEW TRAP. "So `renv` guarantees reproducibility?" No — it pins the R packages. It does not pin R, the OS, or system libraries, and it depends on the sources still being fetchable. Saying "renv plus a repository snapshot plus, for high-stakes work, a container" is the complete answer; saying "renv" alone is the answer of someone who has read one blog post.

Q28. What's your environment strategy across a portfolio, and why is "install the latest" a compliance problem?

SHORT ANSWER. A curated internal repository with dated snapshots as the default source for everything, `renv` per study to pin against those snapshots, and containers for studies where the OS layer matters or where the archive requirement is strongest. "Install the latest" is a compliance problem because it means two runs of the same validated code on different days can differ, which breaks the reproducibility claim your submission rests on.

FOR THE SAS PROGRAMMER. You never let a study float onto whatever SAS version happened to be installed. You fixed the version, you qualified it, and you changed it deliberately between studies, not during one. Same principle, applied to a much larger and faster-moving dependency set.

DETAIL. The layered strategy, from lightest to heaviest:

- **Curated repository with snapshots.** Posit Package Manager is the common commercial choice: it mirrors and curates CRAN, Bioconductor and internal packages, and serves dated snapshot URLs so a study resolves to exactly the package set that existed on a chosen date. It also blocks the "a package disappeared from CRAN" failure. Practitioners commonly assign snapshot management to IT and note that it substantially cuts per-team maintenance — at the cost that migrating a snapshot forces users to absorb many version updates at once.
- `renv` **per study**, pinned against the snapshot. Gives per-study precision inside the org-wide guarantee.
- **Containers** for high-stakes work — an image tagged with the study and the lock date, archived with the submission package. This is the only level that captures the OS and system libraries.
- **Platform qualification underneath all of it.** Posit publishes install-verification documentation for Workbench, Connect and Package Manager, and validation documentation for tidyverse, r-lib, gt, shiny and rmarkdown, which you can leverage as supplier evidence rather than re-deriving — precisely the GAMP 5 second edition "leverage supplier evidence" principle.

Why "latest" fails, said plainly: reproducibility is a claim you make to a regulator. If the environment is defined as "whatever the repository serves today," you cannot make that claim honestly, and the first time you're asked to reproduce a two-year-old table you will find out.

```
# Point a study at a dated snapshot rather than live CRAN
options(repos = c(
  CRAN = "https://packagemanager.example.com/cran/2026-01-15"
))
# every install.packages() and renv::restore() now resolves to that date
```

INTERVIEW TRAP. "Why not just always update — surely newer is safer?" Newer is safer for security and worse for reproducibility, and in clinical analysis reproducibility usually wins. The mature answer is a scheduled snapshot cadence — for example twice a year — with studies adopting a snapshot at start and holding it to lock. Not never-update. Update on a controlled schedule, with studies pinned across the boundary.

Q29. How do you decide which R packages are acceptable for regulatory work?

SHORT ANSWER. A risk-based assessment, not a binary approved list. The R Validation Hub's white paper A Risk-based Approach for Assessing R Package Accuracy within a Validated Infrastructure is the industry reference, and `{riskmetric}` operationalises it — scoring packages on maintenance signals, test coverage, documentation and community health. High-risk uses get more evidence; low-risk uses get less.

FOR THE SAS PROGRAMMER. Closest to how you treated the global macro library versus a one-off utility macro. You did not validate every macro to the same depth; you scaled the effort to what the macro touched. Same instinct, formalised.

DETAIL. How the assessment actually runs:

1. **Score the package.** `{riskmetric}` gathers signals — has a maintainer, has a bug tracker, test coverage, release cadence, downloads, source control, documentation completeness — and produces a risk score. `{risk.assessr}` (Cytel) is a more recent open-source alternative in the same space.

2. **Weight by context of use.** A plotting package that renders a figure and a package that computes a p-value in a primary endpoint analysis carry very different consequences of error. Assess against the use, not in the abstract.
3. **Add your own testing where risk warrants it.** For high-risk functions, run your own test suite against known results — the same logic as qualifying a statistical procedure.
4. **Pin and record.** The assessment is only valid for the version assessed. This is why the whole thing collapses without Q28's snapshot discipline.

Be honest about the state of play, because it is a differentiator: **R PACKAGE VALIDATION IS NOT A SOLVED PROBLEM, AND IT IS NOT STANDARDISED ACROSS THE INDUSTRY.** Every company is doing a version of the same expensive, repetitive work. The R Validation Hub provides guidance and tooling but not a shared, accepted validation package set. As of 2026 there are active attempts to change that — entimo's ValoRpacks aiming at self-service risk-based package validation for sponsors and CROs, and Cytel's `{risk.assessr}` — but nothing has become the industry answer yet. A candidate who says "R packages are validated by the R Validation Hub" is wrong; a candidate who says "the Hub gives us the framework, the validation work is still company-specific and genuinely costly, and that's an unsolved problem the industry is actively working" is credible.

The reassuring counterweight to have ready: the R Consortium's R Submissions Working Group has run pilots submitting R-based analysis packages to FDA, deliberately using a range of open-source packages and formats, and FDA staff successfully reproduced the results — Pilot 3 among them. It is settled that R is submittable. What is not settled is a shared validation approach for the packages.

INTERVIEW TRAP. "Which packages would you approve?" Do not recite a list. The answer is the process, plus the observation that pharmaverse packages (`admiral`, `rtables`, `tern`, `gt`, `xportr`) come with more industry scrutiny and clinical intent than general CRAN packages, which lowers but does not eliminate your assessment burden. Reciting a list is a junior answer to a governance question.

Section B4 — Standardization at portfolio scale, and owning validation governance

Q30. How do you keep forty studies consistent?

SHORT ANSWER. Standards as versioned software packages that studies depend on, a template repository that every new study starts from, and visibility over which study is on which version. Not a document library, and not a shared folder of programs people copy — copies drift the moment they're made, and you can never find them all again.

FOR THE SAS PROGRAMMER. You have lived both models. The global macro library that studies called worked. The "here's the standard program, copy it into your study and adapt it" model always produced forty divergent variants and no way to fix a bug once. The difference between the two is dependency versus copy, and R makes dependency much easier than SAS ever did.

DETAIL. The architecture:

- **Internal standards packages**, versioned semantically, released like software, with their own tests, documentation and change control. Typically split by concern: ADaM derivation helpers, TLF templates, and organisation-specific utilities.
- **A template repository** that seeds every new study with the folder structure, the `renv` setup pointing at the current snapshot, the PR template, the `.gitignore` and the CI configuration. New studies inherit the standard by construction rather than by instruction.
- **A dependency dashboard** — which study is pinned to which version of which standard. This is the governance artefact that makes drift a managed quantity instead of a surprise at audit.
- **A standards governance forum** with a real decision-maker, meeting on a cadence, owning the backlog and the release notes. Without a named owner, standards packages rot within two studies.

The important framing for a Principal/AD interview: consistency is not achieved by mandating that everyone be on the current version. It is achieved by making the current version the easiest thing to start from, making divergence visible, and making upgrades cheap. Any governance model that depends on forty study teams voluntarily doing extra work will fail, and saying so demonstrates that you have actually run one.

INTERVIEW TRAP. "How do you enforce it?" Enforcement through CI, not through email — a check in the pipeline that fails if the study isn't using an approved standards version or an approved snapshot is worth more than any number of SOP statements. But also be honest that enforcement has limits, and the real lever is that the standard has to be genuinely better than what a study team would write themselves. If it isn't, they will route around it and you will never know.

Q31. A standards package needs a breaking change mid-programme. What do you do?

SHORT ANSWER. Nothing is forced onto a study that is already in flight. Studies are pinned to a version, so a new release changes nothing until a study chooses to adopt it. The decision is per study, made by the study statistician and lead programmer, and it is a documented change with its own impact assessment.

FOR THE SAS PROGRAMMER. Exactly the discussion you have had about updating the global macro library mid-study. The right answer has always been "not during a study unless there's a reason," and pinning makes that the default rather than a promise.

DETAIL. The mechanics that make this manageable:

- **Semantic versioning with meaning.** Major = breaking, minor = additive, patch = fix. Studies can then take patches confidently and majors deliberately.
- **Deprecate, don't delete.** A changed function keeps working with a warning for at least one major cycle. Never remove behaviour a locked study depends on.
- **Impact assessment for adoption**, sized to the change. Does it touch a derivation in this study? Then it needs a re-run and a comparison, not a version bump.
- **A bug is different from an enhancement.** If the standard produced a wrong result, this is not a version-strategy question, it is a quality event: identify every affected study through the dependency dashboard, assess impact per study including locked ones, and follow the deviation process. Being able to answer "which studies used the broken version" in minutes rather than weeks is the entire commercial case for the dashboard, and it is the argument that funds this work.

INTERVIEW TRAP. "What if a study is locked and the standard had a bug?" This is the real question, and it is where the interviewer finds out whether you have owned this. You cannot un-lock the study casually. You assess impact on the reported results, and if the results are affected you go through the deviation and, potentially, the regulatory-notification path. What you never do is quietly fix the standard and hope. Say that plainly.

Q32. How do you migrate a team from SAS to R without stopping delivery?

SHORT ANSWER. Run them in parallel and migrate at natural boundaries — new studies start in R, in-flight studies finish in SAS. Migrate the standards first, not the studies. And be honest that the first R study costs more, not less, so the business case has to be built on studies three onward.

FOR THE SAS PROGRAMMER. You have managed a transition before. The failure mode is the same: a mandate with a date and no working exemplar, which produces a team that is bad at both tools simultaneously.

DETAIL. The sequence that works:

1. **Migrate standards, not studies.** Build the R standards packages and the template repo first, so the first R study has something to stand on. This is also where an AI assistant genuinely earns its keep — SAS-to-R translation of a standards library has a natural oracle, because the SAS output already exists and can be compared numerically.
2. **New studies start in R.** Never convert an in-flight study; there is no upside and considerable risk.
3. **Dual-run one real study.** Produce a study's key outputs in both, compare numerically, and fix the differences. This is expensive and it is the thing that buys organisational confidence — nothing else does.
4. **Invest in the environment before the people.** Nothing burns goodwill faster than a programmer spending a day fighting package installation. Get Workbench/Package Manager or the equivalent working first.
5. **Pair, don't train.** A week of classroom R produces very little. A senior SAS programmer paired with an R-fluent programmer on a real deliverable produces a competent R programmer in about a quarter.

6. **Keep SAS.** You will have SAS in the estate for years, and pretending otherwise makes people defensive. The goal is capability, not purity.

Be candid about the timeline: for a team of experienced clinical programmers, real productivity in R takes something like six to twelve months, and the organisation-level payoff arrives later than any business case usually claims. Saying this makes you more credible, not less — the interviewer has almost certainly watched an over-promised migration.

INTERVIEW TRAP. "What's your biggest risk?" Not skills. It is that the R deliverables and the SAS deliverables disagree on a number, and nobody has decided in advance which one is right. Establish that adjudication rule before the dual-run starts — recompute independently from the ADaM data rather than deciding by seniority — or the parallel run generates arguments instead of confidence.

Q33. At this level you'd own validation governance, not execute QC. What's the difference, and what does the framework look like?

SHORT ANSWER. Executing QC means checking this deliverable. Owning governance means designing the system that makes every deliverable checkable, staffing and funding it, monitoring whether it is actually working, and being answerable when it isn't. The distinction is that a governance owner is accountable for the failure rate, not for any individual output.

FOR THE SAS PROGRAMMER. You crossed this line already when you started managing CROs. You stopped checking every table and started asking whether the CRO's process would reliably produce correct tables, and what evidence you had. That is exactly the shift, applied now to your own organisation and its tooling.

DETAIL. What a framework actually contains — six components, and if you can name them fluently you will sound like someone who has built one:

1. **Risk classification.** Not everything gets the same rigour. Primary endpoint derivations, safety outputs and anything supporting a label claim sit at the top; exploratory listings do not. Written down, applied consistently, and defensible.
2. **Defined control set per risk tier.** For each tier: what QC is required (independent programming, review, automated checks), who may perform it, what evidence is retained. This is where double programming, PR review and CI checks each get a defined role instead of overlapping vaguely.
3. **Environment and tooling qualification.** Platform qualification, package risk assessment (Q29), snapshot policy (Q28). Governance owns this so that studies don't each solve it.
4. **Change control.** For standards packages, for environments, for the framework itself. Including the "a standard had a bug and studies are locked" path (Q31).
5. **Monitoring, which is the part people miss.** Metrics that tell you whether the controls work: defect escape rate, findings by tier, how often a QC comparison finds a real difference. A control nobody measures is a control you are only assuming.
6. **Periodic review and honest revision.** Including retiring controls that have never caught anything, which is unpopular and is the mark of someone who is governing rather than accumulating.

The sentence to have ready: my job is to be able to answer, with evidence, the question "how do you know your outputs are right?" — for the portfolio, not for one table.

INTERVIEW TRAP. "Give me an example of a control you'd remove." Have one. A good candidate: 100% independent double programming of every listing regardless of risk, which consumes a large share of QC capacity and, in most organisations' own defect data, catches almost nothing — while the capacity

it frees can go to the derivations that actually carry risk. The willingness to reduce control where the data supports it, and to say you'd want the data first, is the clearest available signal of governance seniority rather than compliance reflex.

Q34. An inspector asks you to reproduce a table from a submission made two years ago. Walk me through it.

SHORT ANSWER. Check out the submission tag from the study repository, restore the pinned environment from `renv.lock` against the archived repository snapshot or the archived container image, point it at the archived frozen data extract, and run. Then show the tag, the lockfile, the extract checksum, the QC record and the pull request that approved the code. The reproduction is the demonstration; the records are the evidence.

FOR THE SAS PROGRAMMER. Same inspection scenario you have handled, with better answers available. Where you used to produce a program listing, a log and a QC form from a document system, you now produce an executable reconstruction plus a complete, timestamped change history.

DETAIL. What you must have arranged in advance for this to work — an inspection is a test of decisions made two years earlier:

```
Archived with the submission, per study:
  - Git repo (full history) + the submission tag        -> the code
  - renv.lock                                           -> package versions
  - the repository snapshot, or a container image       -> the sources / OS
  - the frozen data extract + checksum                  -> the data
  - PR history: who changed what, who reviewed, when    -> change control
  - QC records: independent programming + comparison    -> the verification
  - package risk assessments as of that date            -> tooling qualification
  - environment/platform qualification records          -> infrastructure
```

The two failure points in practice: the sources are gone (a package version archived off CRAN and no snapshot retained — this is why Q28 matters, and it is a real problem, not a theoretical one), and the data extract was never frozen with an identifier so nobody can reconstruct what was actually read. Both are decisions made at lock, and both are invisible until the day you need them.

Have the honest caveat ready too: bit-for-bit identical reproduction is realistic when you archived a container, and merely very likely when you archived a lockfile and a snapshot. If a difference appears, the professional answer is to investigate and explain it — a rounding difference from a numerical library change is explicable; a different subject count is not.

INTERVIEW TRAP. "What if you can't reproduce it exactly?" Never claim you always can. Say what you would have archived to make it as close to certain as achievable, say that a container is the only level that really guarantees it, and say that you would investigate and document any difference rather than characterise it away. Overclaiming reproducibility to an inspector is how a routine question becomes a finding.

Appendix A — SAS to R translation card

The fastest way to read R when you think in SAS. Skim this before an interview; it converts most "how would you do X in R" questions into something you already know.

Data manipulation

You want	SAS	R (dplyr)	R (data.table)
Keep rows	`where` / `if`	`filter(df, x > 5)`	`dt[x > 5]`
Keep columns	`keep=`	`select(df, a, b)`	`dt[, .(a, b)]`
New column	assignment in DATA step	`mutate(df, y = x * 2)`	`dt[, y := x * 2]`
Sort	`PROC SORT`	`arrange(df, a, desc(b))`	`setorder(dt, a, -b)`
Dedupe	`nodupkey`	`distinct(df, id, .keep_all = TRUE)`	`unique(dt, by = "id")`
Group summary	`PROC MEANS` / `BY`	`group_by() |> summarise()`	`dt[, .(m = mean(x)), by = id]`
Conditional	`IF/ELSE IF`	`case_when()`	`fcase()`
Transpose	`PROC TRANSPOSE`	`pivot_longer()` / `pivot_wider()`	`melt()` / `dcast()`
First/last in group	`first.` / `last.`	`slice_head(n=1)` / `slice_tail(n=1)` by group	`dt[, .SD[1], by = id]`
Carry value down	`RETAIN`	`tidyr::fill()`	`nafill()`
Counts	`PROC FREQ`	`count(df, trt)`	`dt[, .N, by = trt]`

Joins — where the mental model breaks

SAS	R	Note
`MERGE a b; BY id;` (both in)	`inner_join(a, b, by = "id")`	
`MERGE a (in=x) b; BY id; if x;`	`left_join(a, b, by = "id")`	
`MERGE` with `if x and not y`	`anti_join(a, b, by = "id")`	
Subsetting merge, keep a's rows only	`semi_join(a, b, by = "id")`	no columns added
Many-to-many `MERGE`	`left_join(..., relationship = "many-to-many")`	**R warns, SAS does not**

THE SINGLE MOST IMPORTANT LINE IN THIS APPENDIX. A SAS `MERGE` on a many-to-many BY group produces silently wrong output. dplyr 1.1+ warns unless you declare `relationship = "many-to-many"`. That warning is a feature — never silence it without understanding why the relationship is not one-to-many.

Missing values

Concept	SAS	R
Numeric missing	`.`	`NA` / `NA_real_`
Character missing	`""` (empty string)	`NA_character_` — **an empty string is NOT missing in R**
Test for missing	`if x = .`	`is.na(x)` — never `x == NA`
Missing in comparison	sorts low, `. < 0` is true	`NA` propagates: `NA < 0` is `NA`, not `TRUE`
Ignore in stats	automatic	`mean(x, na.rm = TRUE)` — **not** automatic

Macros → functions

SAS	R
`%LET x = 5;`	`x <- 5`
`&x`	`x` (no dereference syntax; there are no macro variables)
`%MACRO f(a, b=2); ... %MEND;`	`f <- function(a, b = 2) { ... }`
`%IF` / `%DO` (compile-time)	ordinary `if` / `for` (there is no separate macro pass)
`CALL SYMPUTX`	just return a value, or assign it
Macro library, `%INCLUDE`	a **package** — see Module 3

THE CONCEPTUAL JUMP. SAS macro is text substitution that runs before the code does. R has no such layer: functions are values, evaluated at run time. This is why R has no `&`-style dereferencing, and why "how do I do a macro variable in R" is the wrong question — the right one is "what should this function take as an argument and return".

Appendix B — The 30-day study plan

Daily listening plus a focused reading block. If you only have one week, do days 1, 3, 5, 8, 12, 20 and 30.

Days	Listen	Read	Do
1–5	Orientation + Module 1 tracks	Module 1 Q&A	Install R. Rebuild one real ADSL derivation in dplyr
6–10	Module 2 tracks	Module 2 Q&A	Do one admiral derivation end to end; compare to your SAS output
11–15	Module 3 tracks	Module 3 package sections	Create a package skeleton, write three tests, run R CMD check
16–20	Module 3 tracks again	Module 3 validation sections	Write, in your own words, one page on how you would validate an internal package
21–25	Module 4 Part A	Module 4 Shiny	Build a two-input Shiny app, then split it into a module
26–30	Module 4 Part B + Module 1 again	Appendix A + your weakest module	Rehearse the leadership answers out loud with your own real examples

THE RULE FOR THE LAST WEEK. Stop adding material. Re-listen to what you already have. Recall under pressure comes from repetition, not coverage.

Appendix C — Honest framing

Three sentences to keep in your pocket, because at Principal/Director level the interviewer is assessing judgement more than syntax.

1. **Mark the boundary of your experience.** "I have built that in SAS and read the R implementation; I have not run it on a live study" is a senior answer. Bluffing is not.
2. **Never claim software is validated.** Say what is actually true: the environment is qualified, the package carries a documented test and risk assessment, and the output is verified. See Module 3 — this distinction is the most commonly failed question in the whole set.
3. **Do not invent a story.** Every behavioural answer should come from something you actually did. The structure is in Module 4; the content has to be yours.

Appendix D — What this pack does not cover

Stated plainly so you are not surprised in the room.

- **Deep statistics.** Modelling theory, estimands beyond the programming implications, and survival methodology are out of scope. This is a programming pack.
- **Bayesian / simulation work**, unless a specific role names it.
- **Non-clinical R** — general data science, ML, tidymodels.
- **SAS itself.** Your existing preparation covers that; nothing here re-teaches it.
- **Any claim that you have shipped an R-based submission.** If you have not, the pack teaches you what the pilots demonstrated and where the real limits are — that is a legitimate, defensible answer, and a very different one from claiming the experience.