Supplemental Materials

The purpose of this supplemental document is twofold. First, we briefly describe the two model scripts ("shared_knowledge_model_basic.R" and "shared_knowledge_model_intermediate.R") provided as supplemental downloads which provide an entire code base of the example shared team knowledge development model described in the main manuscript. Both scripts are heavily commented to describe what is occurring in each line of code and therefore we do not cover their operations in detail here. Furthermore, both versions represent the same team knowledge-building process summarized in the main manuscript and lead to the same conclusions. However, each script contains important differences in how the model code is written and highlight how different approaches to model building can leverage certain efficiencies and approaches to model programming in R. We also provide a third script ("shared_knowledge_model_simulation.R") for download; this script is identical to the two primary model scripts but also includes code that (a) demonstrates how to prepare and run a simulation with the example model and (b) reproduce the type of visualizations shown in Figures 7 and 8 of the main manuscript. This script allows one to run the same simulation design described in the section *What Should I do with My Model After it is Built?* of the main manuscript. General details and instruction on writing the code to run a simulation is provided in a later subsection of this supplemental document (*How do I Run Simulations with a Model?*).

The second purpose of this supplemental document is to describe several additional considerations relevant to building computational models as well as introduce some more intermediate programming concepts not covered in the main manuscript, but which readers may find useful as they gain proficiency building computational models. As this material is intended to accompany the material presented in the main manuscript, it is most relevant to understanding how to build and work with computational models using the R programming language.

Full Model Scripts for Example Team Knowledge-Building Model

Two primary model scripts are available for download that implement the full example team knowledge-building model introduced in the main manuscript. The first script is named "shared_knowledge_model_basic". This version is intended to be more "beginner-friendly" by building upon and closely aligning with the code introduced in the main manuscript for the example model (i.e., Figure 6). This version also only implements the programming concepts discussed in the main manuscript (i.e., loops, if-statements, etc.) to represent the full computational process theory. Consequently, it should be readily interpretable based on the content covered in the main manuscript.

The second script is named “shared_knowledge_model_intermediate”. This version introduces several changes to the programming commands and structure of the example computational model but not its overall functioning. We elected to provide this secondary version of the model script for two reasons. First, it showcases an alternative approach to writing model code in R that relies on vectorized operations. Vectorized operations are discussed further in a later subsection of the supplemental materials (*How Can I Improve the Speed and Computational Efficiency of Model Code?*), but the basic premise is that vectorized operations provide an alternative to writing loops to repeatedly execute *simultaneous* computations. Using loops is often a more intuitive approach to programming and perfectly acceptable in model code. However, they have the potential to be less computationally efficient in R, and there are many times in which a vectorized approach can be employed to achieve the desired result. In the beginner-friendly code, for-loops were used to perform a variety of tasks in both the model initialization and model algorithm sections. In the intermediate code, all for-loops have been removed and replaced by vectorized operations. For example, a for-loop was used in the model initialization section of the “shared_knowledge_model_basic” model script (line 174) to loop through each piece of information one at time and sum how many agents initially knew that information piece. In contrast, this same task is performed in the “shared_knowledge_model_intermediate” model script (line 169) using the vectorized `colSums()`operation provided in base R. However, the biggest difference between the vectorized and non-vectorized scripts can be seen in how the learning action/event is implemented for agents. In the beginner-friendly code, a for-loop is used to check whether agents attended to the shared information and updated their knowledge pools one agent at a time (line 263). In the intermediate code, this process is carried out using a vectorized operation that effectively performs the attention and learning actions/events for all team members simultaneously and in parallel (line 255). In this model, both versions produce the same result because the order in which agents attend/learn to information during each model iteration is inconsequential.

A second but related purpose of providing the intermediate version of this model code is to demonstrate the inherent equifinality involved when coding computational models. There are often many functionally equivalent means of executing the same substantive process via a model script. In most cases, these differences come down to programmer preferences/comfort, considerations of computational efficiency, and how closely one attempts to match the “behavior” produced by the computer code with the substantive process being represented. For example, using a

vectorized approach to implement team members learning is (marginally) more computationally efficient than using a for-loop in R. However, one could also argue that it is more conceptually aligned with how the real phenomenon occurs in that it represents team members learning in parallel when a piece of information is shared rather than each member learning one after the other (as a for-loop implies). In the end, it is most critical to ensure that the code written for a computational model faithfully reproduces the intended computational process theory, and the outcomes it generates are interpretable. After those goals have been achieved, refining the code to improve readability/complexity, fidelity, and computational efficiency can be pursued.

Additional Considerations for Building and Working with Computational Models

The main manuscript summarizes what we consider the most fundamental conceptual points and programming concepts for learning to build computational models to represent theory. However, there are many other useful techniques and important skills to be mastered which improve proficiency at writing and using computational models. In the sections below, we summarize five additional considerations and recommendations that cover more "intermediate" programming and model-building topics than those covered in the main manuscript. We recommend this material for readers who have a comfortable grasp of the concepts discussed in the main manuscript (i.e., can write functions, loops, and if-statements to represent how an action/event proposed in a computational process theory occurs) and are ready to take the next step towards improving their model-building skills and knowledge.

What Programming Language or Modeling Software Should I Use?

Selecting a programming language or modeling software holds several implications for both learning how to create computational models as well as the procedures used to transform a computational process theory into executable code. For example, different programming languages or modeling software typically use different commands and syntax (i.e., the operators, functions, and associated rules for how those elements can be combined) to instruct a computer how to perform desired model operations, rely on different interfaces to work with a model, its code, and for conducting simulations, and provide different capabilities for writing, debugging, visualizing, running, exporting, and analyzing a computational model and the data it produces.

The main manuscript summarized much of our rationale for describing how to build computational models in R, but there are many other programming languages and software applications available for writing computational

models (see Fioretti, 2012, Weinhardt and Vancouver, 2012, or Macal & North, 2010, for introductory descriptions of several modeling frameworks of relevance to social and organizational scientists). At the risk of oversimplification, these can be organized along a continuum from more general-purpose programming languages/applications (e.g., JavaScript, MATLAB, Python, R) to more purpose-built software designed for developing specific types of computational models (e.g., NetLogo, Vensim, AnyLogic). Given their broad application, general-purpose programming languages often do not provide features directly tailored towards coding and simulating computational models. However, this lack of specialized functionality is compensated for by the potential to create virtually any type of model for any type of phenomenon all within a single programming language. Additionally, although specifics of the programming syntax often differ between general-purpose languages, many rely on similar underlying coding philosophies, operations, functionalities, and data types/structures. Thus, there is often a relatively high degree of knowledge transferability across languages (i.e., easier to understand or borrow from code written in different languages). In comparison, more purpose-built modeling frameworks include features and functionalities intended to facilitate several aspects of programming and simulating computational models. For example, many purpose-built modeling frameworks attempt to use more intuitive syntax and techniques for organizing code (e.g., "what you see is what you get" [WYSIWYG] programming), offer options for building models using a graphical interface, and provide custom integrated development environments (IDEs; software and associated bundle of features provided for working with code) with tools aimed at streamlining certain modeling and simulation tasks (i.e., creating static or dynamic visualizations, creating simulation conditions, model-fitting/parameter estimation). However, this tailored functionality typically comes at the cost of a restricted set of model types that can be implemented within a given application and comparatively lower knowledge transferability across languages/applications (i.e., more difficult to understand or borrow from code written in different languages).

Despite these differences, we believe the decision regarding which programming language or modeling software to learn and use for building computational models is less consequential than it may appear. Learning to construct, program, simulate, and draw inferences using computational models requires significant investment irrespective of what language or modeling framework is chosen. For example, the extent to which the "user-friendliness" of a purpose-built modeling framework versus the flexibility of a general-purpose programming language is desirable is not always readily apparent. Additionally, the advantages or disadvantages of a particular platform may

be unclear until one begins to build a model and often depends on a host of factors (e.g., intended use of the model, goals of the researcher, norms and preferences within specific research and practice communities, comfort with syntax-based versus "point-and-click" user interfaces, knowledge of different types of computational models, their uses, and their limitations). For example, although learning/teaching with a more purpose-built platform may remove certain barriers to entry, we have observed that learning to write model code using a more general-purpose language (such as R) can sometimes be highly beneficial for sharpening the critical and process-oriented thinking skills imperative when building computational models. With fewer built-in features for streamlining the model-building process compared to bespoke modeling software, programming a model in a general-purpose language forces a researcher to consider exactly how the initial conditions and mechanisms of a theory will be represented, carefully operationalize those decisions into code, and evaluate whether/how those choices are consistent with the logic of the explanatory account.[1] Consequently, using a more general-purpose architecture to code a model invites one to approach model development "more like a programmer" (e.g., Chambers, 2000)—which in our experience can benefit the development of process theory.

In sum, our typical response to the question of which programming language/modeling software to work with is that it does not matter greatly. Pick an option that is appealing, commonly used by others building computational models in one's area, and/or for which accessible resources are available to assist with coding and model-building. Additionally, learning and switching between different programming languages or modeling software typically becomes easier as one begins to gain confidence and experience with the fundamental competencies of model-building (and can even facilitate acquiring these skills)—so the most important choice is simply to make a choice!

Why is my Model Function not Working as Intended?

Significant time and attention will be devoted to creating and using functions to represent theory when writing computational model code. Understanding how to effectively construct and implement functions is therefore a critical learning objective for gaining model building proficiency. Although the main manuscript presents the fundamentals for working with functions in R to control how actions/events occur in a computational model, there are several rules and behaviors specific to functions in R that can make this activity more intuitive, assist with troubleshooting functions that are not operating as intended, and improve coding efficiency. To make these principles

more concrete, we will use the following function as an example. This function takes three arguments as inputs, performs a series of calculations, and then returns a single result:

```
# Code for custom function
some_function <- function(arg1, arg2, arg3) {
      out1 <- arg1 + arg2
      out2 <- out1 / arg3
      out3 <- out1 * out2
      out4 <- out3 - var_x
      return(out4)
}
```

As noted in the main manuscript, running these lines of code in R will produce a new function object and bind it to the name `some_function`. Once these lines of code are executed, this function can be used like any other programming command by calling the function, providing the input arguments, and executing the command (e.g., `some_function(arg1 = 5, arg2 = 10, arg3 = 3)`).

A simple but easily overlooked principle when first learning to work with functions in R is that generally a name should always be bound to them except under special circumstances (e.g., writing an anonymous function inside other functions). Additionally, the code used to bind a name to a function must be run before a function can be used. In the provided example, the code used to define the function begins with the `function(arg1, arg2, arg3)` command and includes all the lines of code within the opening and closing curly braces `{}`. A name (`some_function`) is then bound to this function using the assignment operator (`<-`) that allows it to be stored in the computer's memory and called from the model script using the bound name. Failing to bind a name to a function in this manner renders the function unusable in subsequent code.

Another important principle concerning functions in R is how the value of variables referenced within a function are identified. Functions built in R may contain three types of variables (Venables et al., 2022): formal parameters, local variables, and free variables. *Formal parameters* are those included in the argument list of a function (e.g., `arg1`, `arg2`, and `arg3` from the code used to create `some_function()`). The values for formal parameters are determined by directly associating each argument with a value when the function is called (e.g.,

`some_function(arg1 = 5, arg2 = 10, arg3 = 3)` binds each of these formal parameters with the specified value). *Local variables* are those whose values are determined inside the body of a function (i.e., between the `{}`), usually by evaluating an expression. For example, the values associated with `out1`, `out2`, `out3`, and `out4` are each determined as part of the computations performed within the body of `some_function()` when it is run. Importantly, the values associated with both the formal parameters and local variables of a function are considered internal to that function. Effectively, this means that these objects do not exist, nor can their associated values be referenced, outside of the function's body. The `out1` object created in the first line of the `some_function()` command thus only exists temporarily while the function code is running; afterwards, it is erased from memory and can no longer be accessed.

Any other variable that exists as part of a function in R but not defined as either a formal parameter or a local variable is considered a *free variable*. In the `some_function()` code, the variable `var_x` included in the line of code used to compute `out4` is a free variable because its value is not defined by either the arguments or operations of the function. The values associated with free variables must instead be determined by looking "outside" of the function. The procedure used in R to identify free variables is called *lexical scoping*. Lexical scoping means that the values associated with a free variable are initially searched for in the environment where the function was created. An *environment* in R is simply a collection of objects (e.g., label-value pairs created using the assignment operator; running `var_x <- 1.5` associates the label `var_x` with the value `1.5` and stores this label-value pair in an environment). R allows for multiple environments to exist at any one time, but the most important and commonly used is the global environment (i.e., workspace) created each time a session in R is opened. Creating multiple environments in R is a relatively advanced technique and is mostly unnecessary for building computational models. Consequently, it is generally sufficient to rely on the global environment for writing model code such that all variables, objects, and functions reside in the default workspace.

Returning to lexical scoping, assuming a model function is created in the global environment, R will attempt to search for the values of any free variables found in that function in the global environment when the function is run. In the context of the example code for `some_function()`, this means that R would attempt to identify whether an object named `var_x` exists in the global environment in order to determine its value. If an object with that label is found, its associated value will be used for `var_x` to perform the desired operations inside the function. If no object

with that label is found, an error will be thrown indicating that `var_x` is undefined. Thus, attempting to use the `some_function()` command will not work unless a value for `var_x` has been defined outside the function:

```
# Does not work because var_x is undefined
test_1 <- some_function(arg1 = 5, arg2 = 10, arg3 = 3)

# Works because var_x is defined
var_x <- 1.5
test_1 <- some_function(arg1 = 5, arg2 = 10, arg3 = 3)
```

Lexical scoping and how the different types of variables used in a function are characterized are unique aspects of R programming and can afford some useful computational advantages. However, these behaviors can also be a source of confusion and produce unanticipated outcomes if one is not aware of how these principles operate. Given these considerations, we offer two recommendations when writing model functions in R to control how actions/events occur in a computational model:

1. **Use free variables sparingly — or ideally not at all**. Although using free variables in a model function can be an efficient and useful coding technique, they can easily lead to coding errors/mistakes that result in unintended behaviors. For example, a free variable defined in the global environment may be unexpectedly overwritten or set to an unexpected value prior to being used in a function; alternatively, a free variable may be defined in the global environment with the intention of using it in a function, but a local variable with the same name is accidentally included in the function and thus used instead. If a model function is created with the intention of using a variable/object that is defined outside a function's local scope (e.g., function computations that rely on a model parameter or an object created during model initialization), an additional formal parameter can be included in the argument list of the function to ensure that variable is explicitly passed into the function. For example, our prior function could be rewritten as:

```
# Add formal parameter for var_x defined outside of the function
some_function <- function(arg1, arg2, arg3, var_x) {
      out1 <- arg1 + arg2
      out2 <- out1 / arg3
```

```
    out3 <- out1 * out2
    out4 <- out3 - var_x
    return(out4)
}

# Define new formal parameter with existing object
var_x <- 1.5
test_1 <- some_function(arg1 = 5, arg2 = 10, arg3 = 3, var_x = var_x)
```

2. **Be mindful of the desired outputs returned from a function**. In the main manuscript body, we noted that the `return()` command is used to specify what data/object is the desired output of a function. The present considerations provide further context as to why declaring a return statement is important. All objects created inside a function are considered local variables and thus are removed after the function has finished executing. However, a return statement instructs R to immediately execute a single expression referenced inside a `return()` command in the environment from which the function containing that return statement is run. Thus, the line `return(out4)` shown in the previous example instructs R to run the expression `out4`. Because `out4` is a data object, this return statement means that the result we will "see" produced by our function (i.e., its output) will be the `out4` data object. Using a return statement to reference a data object created inside a function in this way thus provides a means to save an otherwise temporary and inaccessible local variable. We also note that R functions can return the last printed object inside a function even without a `return()` statement. In `some_function()`, if we delete `return()` around `out4`, then the function would still return the value of `out4`. Some R programmers like to reserve the use of the `return()` statement only for cases when an "early" return might be possible (e.g., inside of `if()` statements). We consider both options viable as including a `return()` statement makes it clear an output is being produced.

    Additionally, there are many times when it is useful or desirable to return *multiple* data objects created within a function (e.g., return both `out4` and `out3` when `some_function()` is run). Technically, R functions can only return one object. Thus, in order to return *multiple* objects in R functions, it

is necessary to combine these separate objects into a single data object capable of holding multiple values (e.g., a vector, matrix, list) and then return this new object. For example, revising the return statement in the example function to `return(c(out3, out4))` instructs R to create a single vector in which the first element holds the data associated with `out3` and the second element holds the data associated with `out4`. Alternatively, a local variable inside the function could be created first to create the desired structure of the output object (e.g., `final_output <- c(out3, out4)`) that is subsequently referenced in the return statement (e.g., `return(final_output)`). We note that in other cases programmers may want to use the `list()` function as opposed to the `c()` function, since the `c()` function performs a conversion of data structures (e.g., converts a matrix into a vector) whereas the `list()` function creates a data object in which objects with different types of data structures can be stored.

How Should I Initialize and Structure Data Objects in my Model?

Social and organizational scientists are well-trained in measurement and statistical analysis, making them adept at planning for, gathering, and working with empirical data. Working with data in the context of computational model building, though, often entails different considerations than working with traditional empirical data. For example, many statistical analysis software applications expect data to be structured as a two-dimensional data table in which rows represent records and columns represent all the variables collected about those observations. Consequently, social and organizational scientists tend to think about data and data collection with this structure in mind, and there are many tools and resources related to the "data wrangling" operations for coercing data into such formats (e.g., Braun et al., 2018). Although one may end up using this data structure and engaging in similar data wrangling activities when preparing the data generated by a computational model for further analyses, the standard two-dimensional data table and the associated principles for organizing data for statistical inference can often be a less useful or effective way in which to think about working with *data objects* and writing model code to represent/operate on the core processes and mechanisms proposed in a theory. In this section, we thus highlight several points and recommendations for approaching the structure and initialization of data objects when building and working with computational models.

Data Objects and Data Structures

To orient our discussion, it can be useful to consider the meaning of "data" from the perspective of building computational models and writing model code. In social and organizational research, it is common to think of data in terms of the information gathered from or about a focal object/entity (i.e., research participants, situation/environment) that will be used to aid subsequent description, analysis, and reasoning. Although this description can analogously apply to certain aspects of data in a computational model, an alternative perspective more consistent with how programmers think about data is as objects stored and processed digitally on a computer (Pickle, 2022). A subtle yet key distinction between these perspectives concerns the implied "purpose"/goals of data. Empirical researchers generally think of data as a "means to an end." That is, the key purpose of data is to inform descriptive constructs/variables, examine distributions, and to use when conducting statistical analyses. The organization and structure of data into relational/two-dimensional tables that directly link all observations collected with the focal units of analysis is both conducive to and efficient for these activities. In contrast, a programmer may be required to think about data as an "end" in and of itself. That is, all operations written into computer code are built to produce a specific output (i.e., a Boolean TRUE/FALSE value, numerical values, a function) whenever they are enacted by the computer. For example, every line of code in the model parameters, model functions, initialization, and model algorithm sections of the example model scripts in Figures 3-6 of the main manuscript provides an instruction to the computer to conduct an operation that results in the creation of objects (i.e., data) that will be stored in the computer's memory. Some of these objects may be used only for the purposes of performing/informing subsequent tasks in the model code (e.g., the `still_learning` variable shown in the code of Figure 6 is only used to control the iterations of the while-loop and thus carries relatively little analytical value), whereas others may be of interest for subsequent analysis (e.g., the `know_pool` variable records simulated team members' knowledge given the assumptions/parameters of the model and thus may form the basis of analytical data). Thinking through how to organize and structure data objects that facilitate performing operations necessary to enact the logic of a computational model as well as recording observations useful for later analysis is thus a frequent occurrence when programming computational models.

An important recognition (and commonly observed pitfall) when first learning to write model code is that relying on the familiar two-dimensional data table may not always be the most convenient or efficient structure when writing model code. There are many ways in which the data objects used in a computational model can be structured.

The available types, format, and options for working with those data structures will vary depending on the specific programming language or modeling software used to code a model, but there are enough commonalities to consider some first principles for structuring and organizing data objects in model code. In base R, there are five fundamental data structures: the vector, matrix, array, data frame, and list. Describing the specific properties, use cases, and how to work with these different data structures in R is beyond the scope of this document, but more information on these topics is available in Wickham (2019) or Wickham et al. (2023). As summarized in Figure S1, arguably the most critical distinction between these data structures is the dimensionality (i.e., whether data can be organized into one-, two-, or multiple-dimensions that allow more complex relational structures to be maintained) and content they can accommodate (i.e., whether data must all be of the same type or can be different types).

Beyond these characteristics and their implications for which and how data is stored/referenced within a given data structure type, there are almost no concrete or normative rules for deciding which data structures should be used to organize and store the data generated by the operations performed in a computational model. Nevertheless, we offer the following general recommendations when approaching model development and particularly when using a general programming language such as R for model coding:

1. **Consider using unique data objects to refer to unique variables/concepts from a theory.** The structure of the two-dimensional data table common in empirical research is generally such that one dimension refers to a common set of units/observations (i.e., each row of a data table refers to unique research participants, time points, etc.) that can each be described by the same set of variables/concepts referenced in the other dimension of the data table (i.e., each column of a data table refers to a unique scale item, variable, etc., collected from or observed about each focal unit). In this respect, this data structure records/includes data about *multiple* variables/concepts of interest either contained by or descriptive of a specific type of focal unit in a *single* data object. Although such a data structure can be used for organizing and structuring data in model code when similar conditions are met, it may be more convenient to write and interpret code in which unique data objects are used to refer to unique variables/concepts from a theory instead.

   For instance, the example computational model script included in Figure 6 of the main manuscript included code to create and store data on the cognitive ability and conscientiousness levels for each simulated team member. In that code, each of these variables/concepts were stored in separate data

objects (e.g. `cog` and `cons`, respectively). Alternatively, model code could have been written to initialize and store these data into a single data object using any of the available two-dimensional data structures provided by R (e.g., matrix, data frame, array) such that each row referred to a specific simulated individual and the columns referred to the desired variable:

```
# Initialize conscientiousness and cognitive ability levels for all simulated individuals
# into single data frame
agent_attributes_df <- data.frame("cog" = runif(n = n_members, min = .1, max = .33),
                                  "cons" = runif(n = n_members, min = .3, max = 1))
```

Although there are potential advantages to writing model code that uses fewer data objects to reference variables/concepts from a theory, there are also tradeoffs. Referring to data in a model script requires writing code that appropriately points to the correct data object and the location within that data object indicating where the desired data is stored. For example, referencing the cognitive ability level of all agents in R using the single data object structure shown above would require writing `agent_attributes_df$cog` in model code versus simply `cog` if this variable were stored in its own unique data object. Though this seems a relatively trivial change in the present example, other models may necessitate using more complex data structures (e.g., nested lists, multidimensional arrays) to initialize/record data that would require even more complex data referencing. This has the potential to make model code more difficult to work with and more prone to coding errors. Relatedly, using single large data objects to structure multiple variables/concepts can make code less readable and interpretable. In many cases, the lines of code in a computational model script are written to represent a specific action/event formalized using a relatively small number of variables, concepts, attributes, etc. The ability to look at a line of model code and easily see what variables/concepts are being used in a particular operation without needing to unpack a complicated data reference can thus facilitate both writing and reading computational model scripts.

2. **Keep data objects as simple as possible.** R and some other general purpose programming languages permit data objects with highly complex structures to be created and used in model code. For example, although data is often structured in R data frames in a manner consistent with that of the traditional two-

dimensional data table (i.e., each cell stores a single value), R data frames permit different types of data to be stored inside each cell (e.g., each cell stores a vector or matrix of values) provided that all columns consist of the same number of elements. When first learning to write model code, we advise making every effort to create and use data objects in model code that do not rely on complex organizational structures. Specifically, we recommend attempting to structure data objects in a computational model such that each of its elements (i.e., a location where data may be stored and referenced, such as the position in a vector or the cell of a matrix) references only a single value (i.e., number, Boolean value, character string, etc.) whenever possible. Following this data structure principle typically makes model coding easier, less prone to errors, and easier to engage in data wrangling for purposes of further analysis.

Echoing a sentiment from the previous recommendation, following this principle when writing computational model code in R may again mean that the conventional two-dimensional data table is neither the best nor most useful structure for approaching the structure of data objects used in a computational model. For example, the model script shown in Figure 6 of the main manuscript contained a data object called `know_pool` used to record how much each simulated individual knows about each piece of information in the team's environment. That script initialized the `know_pool` object as a two-dimensional matrix with rows equal to the number of simulated team members (`n_members`) and columns equal to the number of information pieces available to learn (`n_info`). Accordingly, the implementation of the `know_pool` data object in that model script is such that *only* the current state of the team's knowledge pool at each model iteration is saved. The data stored in the `know_pool` object is overwritten during each model iteration to reflect only what each member on the team knows at the current time point and thus no record is maintained of how the knowledge held by each individual changes over time. However, it may be desirable to store what the team's shared knowledge pool looked like at each time step for subsequent analysis and thus an alternative data structure is needed.

One way to store this time-varying data might be to attempt to keep the two-dimensional structure of the `know_pool` object but configure it in a "long" format similar to how data is often organized when analyzing multilevel data using random coefficient modeling (e.g., person and time columns are used as identifiers in the data table that enable one to uniquely identify the state of each piece of information for

each person by time combination; see Wickham et al., 2023). However, this structure is both more difficult to initialize (because of the need to organize the person and time identifiers correctly for the data object a priori) and use in the model code (because of the need for more complicated selection/extraction references to choose data from and/or indicate where to save data within this data object). An alternative for structuring this data in R could be to use either a multi-dimensional array or list structure that allows one to save the simple person by information `know_pool` object at each time point (i.e., as a person by information matrix "slice" within a three-dimensional person by information by time array, or as a list in which each element contains the person by information matrix observed at each time point). In sum, the key takeaway of this recommendation is to prioritize simplicity and utility when structuring data objects used for performing operations in model code (i.e., accessing, selecting, and manipulating data is clear and easy).

3. **Consider which data is likely to be useful for analysis while coding and prepare accordingly.** The prior recommendations are intended to facilitate creating and working with data objects necessary to represent the logic specified in a proposed theory into executable computational model code. We believe this is the single most important objective when building a computational model and therefore strongly advise adhering to that guidance. However, it is important to note that the resultant data objects produced by following these recommendations frequently do not result in data structures immediately amenable to subsequent analysis. Although this may seem undesirable or unfortunate, the "goals" of the data structures and corresponding data objects used for conducting analyses versus building a computational model are often different and thus the organization of data used for those purposes naturally reflect those differences (Wickham et al., 2023).

   Consequently, it can be useful to actively monitor which data generated by the operations carried out in a computational model are likely to be of analytical interest and whether subsequent operations will be necessary to make such data useful for analysis while writing code. In the main manuscript text, we noted that many of these activities may be included in a model script *after* the model algorithm code as they constitute configuring data output and are not technically part of building a computational model (see Figure 2 and the section entitled *How Should I Organize Model Code?*). However, there are often places in the initialization and algorithm sections of a computational model script where efforts can be made to (a) ensure

that data useful for exploring interesting questions or probing the assumptions/implications of a model are being generated and (b) incorporate data objects that facilitate examining or visualizing patterns conducive to those goals directly in the model code.

To provide a concrete example, one potential data source of interest that could be recorded from our example model of shared knowledge development in teams might be the trajectory of knowledge growth for members and the team as a whole. Specifically, it may be useful to record data about how many pieces of information are fully known by each team member at each time point as these observations could be used to explore how, say, the distribution of cognitive ability or conscientious levels within a team affects the trajectory of shared knowledge emergence given the assumptions of the proposed theory. We previously noted that this data could be derived from the `know_pool` object if it were changed from its current implementation in Figure 6 to record the team's knowledge pool at each time point. However, the current `know_pool` object also is not configured to record the *exact* data we are interested in analyzing (i.e., the number of information pieces known by each team member) nor would recording the `know_pool` matrix data object at each time point result in a format easily amenable for analysis. As such, it may make sense to consider initializing and updating an additional data object that records this data of interest at each model iteration directly in the model algorithm using a more convenient structure for analysis (e.g., a new time by agent matrix data object that records the total number of information pieces known by each agent at each time point). The complete model scripts we have provided for download as online supplements implement this additional data object in the model code (see the `agent_know` object created on line 212 and how it is updated on line 305 in the "shared_knowledge_model_basic.R" model script). In sum, the data objects used to implement a theory's logic into computational code are often not structured in a manner that accommodates subsequent analyses. Opportunities for satisfying both of these goals should thus be considered throughout the code writing process.

Size and Growth of Data Objects

The previous subsection highlights that an important consideration when building model initialization code concerns planning for how the data generated by a model will be recorded. In many cases, *what* data a model will produce — and therefore what variables should be recorded — during a model run is known in advance, but it may

not be possible to know *how much* data will be produced. This is particularly true when a researcher intends to record data generated over (simulated) time each model iteration. For example, the pseudocode for the example shared knowledge development theory summarized in Table 1 of the main manuscript indicates that the model algorithm should repeat until all members on the team have fully learned all possible information. There are many possible data/variables that could be recorded while this process plays out each model iteration that may be useful for exploring the implications of the proposed theory (e.g., which agent shared information, what information was shared, how many pieces of information each agent knows, etc.). However, one challenge with initializing data objects to store these data is that it is not possible to determine a priori *how much* data will be generated during a given run because it is not possible to know how many model iterations will be performed before the stated stopping criterion is reached.

This circumstance illustrates a dilemma commonly encountered in programming: how should data objects in which the size, structure, etc., can change dynamically be initialized and used in code? In practice, there are two general approaches for handling such situations. The first is to write code capable of "growing" a data object by appending new records to it as necessary (e.g., save the data generated during each model iteration and then successively bind/append those items together to create an increasingly large object that holds all the data). With this method, the data object may initially be very small, but it is expanded on-the-fly and on-demand as data is generated and thus its size adjusts accordingly. A second option is to first initialize an "empty" data object whose size is larger than necessary and then insert data into that object as it is generated (e.g., create a blank matrix with many more rows than are likely to be needed and subsequently write the data produced during each model iteration into successive rows of the matrix). Such data objects are often described as being "pre-allocated" because their size, structure, etc., is reserved in a computer's memory in advance and await data to be recorded into them. Empty space remaining in a pre-allocated object after a model run has completed is typically removed so that the final object contains only usable data (e.g., delete rows of a matrix that do not contain any data after a model is finished running). Although both methods can be implemented in R, using pre-allocated data objects is generally more computationally efficient and is commonly recommended. In fact, the pre-allocation method can be used to initialize both "fixed-sized" or "variable-sized" data objects (i.e., objects in which the total size/amount of data is known versus unknown in advance, respectively).

To provide a more concrete example of these concepts, consider the initialization procedure used to create the know_pool object in the complete code for the team knowledge building model provided in the supplemental materials ("shared_knowledge_model_basic.R"):

```
# Initialize empty knowledge pool object
know_pool <- matrix(0, nrow = n_members, ncol = n_info)


# Populate knowledge pool object with data
for(i in 1:n_members) {
      know_pool[i, ] <- sample(initial_info_state, size = n_info,
      replace = TRUE, prob = initial_info_probs)
}
```

In this case, the overall size of the know_pool object is "fixed" by a set of model parameters (i.e., the matrix object will contain n_members rows and n_info columns). A fixed-size object can therefore be pre-allocated during model initialization to hold the initial knowledge pools for each agent. The example code here uses a for-loop to then populate the object with the desired data. Specifically, a random sampling procedure is used to create a vector of values indicating the pieces of information an agent initially knows, which is then inserted into a single row of the know_pool matrix. This procedure is performed for each row (i.e., agent) in the matrix to generate the initial knowledge pools for simulated members. Note that although the size of this data object (i.e., number of rows and columns) is fixed, the data it stores need not remain constant. Indeed, as agents in the example model learn, the code in the model algorithm updates the data stored in the know_pool object to reflect these changes.

Although the data stored in this fixed-size know_pool object can change over model iterations, the amount of data it contains will always be the same. In contrast, imagine a researcher wished to record which agent spoke and what information piece it shared during each model iteration. One way this data could be structured/organized in R is as a matrix with rows corresponding to the time point/model iteration and two columns for recording which agent spoke and what it shared (respectively) at that time point/iteration. Because it is not possible to know in advance how many time points/iterations will be produced when the model is run, we do not know the exact number of rows of data this matrix must be able to store and therefore the size of the object cannot be exactly

specified at model initialization. The pre-allocation method can thus be used as shown on line 202 of the "shared_knowledge_model_basic.R" script:

```
# Initialize empty object for storing speaking data
spoke <- matrix(NA, nrow = 5000, ncol = 2)
```

Running this code creates a matrix data object named `spoke` containing an arbitrarily large number of rows (5000) and two columns (one to hold data indicating which agent spoke and one to hold data indicating what piece of information was shared). In this example, the matrix is populated entirely with the value `NA` to indicate that no data has yet been generated. Each row of the matrix can thus be seen as a pre-allocated "placeholder" prepared to hold data about agents' information sharing activities generated during a single model iteration (i.e., first row will hold the data generated during the first time point/model iteration, second row the data generated during the second time point/model iteration, etc.). If a given model run concludes before all 5000 rows of the `spoke` matrix are populated (i.e., the model run required fewer than 5000 time points to complete), unused rows can be deleted prior to data analysis. If a model run required more than 5000 iterations to complete in this example, an "out of bounds" error would be encountered that indicates an attempt was made to update data that does not exist (i.e., the computer attempted to record data in the 5001st row of the `spoke` matrix, but only 5000 rows exist). Consequently, it is usually good practice to initialize data objects that are much larger than are likely to be necessary when using the pre-allocation method. This pre-allocated size can later be reduced/calibrated to a smaller "buffer" after gaining familiarity with how much data is typically produced by the model.

How Can I Improve the Speed and Computational Efficiency of Model Code?

The main manuscript noted that loops are one of the main tools for controlling the timing and effect of when actions/events occur in a computational mode. Furthermore, loops afford the capability to model circumstances in which the specific order an action/event occurs across units is either meaningful/important or inconsequential. However, in the latter case in which the order that units experience modeled actions/events or their effects can be ignored, it can be preferable to use *vectorized* operations instead of loops in model code. Unlike a loop that performs code across multiple units one at a time (i.e., a learning function is first run for agent 1, followed by agent 2, followed by agent 3), vectorized operations seek to perform code for multiple units at the same time (i.e., a learning function is run for agents 1, 2, and 3 simultaneously).

Loops are a powerful and versatile technique in programming and model coding; furthermore, they tend to be intuitive to implement once the basic concepts are grasped. However, loops also carry some disadvantages, particularly when using R to write model code. In many ways, the R programming language is designed to reduce the need to write loops to perform repeated operations across units. For a variety of reasons then, R is not particularly well-optimized for heavy usage of loops. That said, the main tradeoff between loops and vectorization tends to be computational efficiency (i.e., how much processing power, time, and internal memory is necessary to run a model), which tends to be a lower priority when building computational models—especially when first learning to code models. Nevertheless, we introduce a few of the tools/commands unique to R that allow for more vectorized-like operations in this section for those interested in taking the next step to improve the efficiency of their computational models in R.

In some instances, implementing vectorized operations can be preferrable to implementing loops because it reduces the amount of code that needs to be written. For example, R performs vectorization every time a mathematical operator (e.g., `+`, `-`, `*`, `/`) is used on a data object containing multiple values (e.g., a vector, matrix, array, data frame). Consider a model in which a variable named `age` is used to record the ages of a group of simulated employees (e.g., `age = c(30, 21, 54, 43, 28)` stores the chronological age of five different individuals). Adding one year to the age of each of these employees using a for-loop would require selecting each value in the `age` vector one at a time, adding one to the selected value, and then replacing the old value with the new value in the `age` vector:

```
# Non-vectorized addition
age = c(30, 21, 54, 43, 28)
for(i in 1:length(age)) {
      age[i] = age[i] + 1
}
```

Alternatively, the code:

```
# Vectorized addition
age = c(30, 21, 54, 43, 28)
age = age + 1
```

produces the same result in R because the operation is effectively performed on all elements in the `age` vector simultaneously. Note, however, that the looping method should still be used if the *specific order* in which individuals aged in this model was conceptually important and/or the model code incorporated mechanisms that affected how actions/events or their associated effects unfolded for other individuals (e.g., the amount of money available to spend on a birthday celebration decreases with each birthday celebrated in a given year such that less money is available to spend on celebrations for individuals with later versus earlier birthdays).

In addition to performing simple vectorized operations, R provides the means to perform vectorized-like operations across units that are more complex or involve multiple steps through its family of *apply* commands. There are several types of apply commands in R, but they all require (1) a function to perform on units and (2) a series of values/objects that serve as input to that function. When writing a computational model in R, the function provided to an apply command often represents the operations used to perform an action/event and the input values typically provide a means for identifying the set of units that will (simultaneously) experience that action/event. Taken together, these components can be combined in an apply command to execute the stated function using each input value/unit simultaneously (i.e., *apply* a function to the specified units at the same time).[2]

The apply commands arguably most useful for controlling the performance of simultaneous actions/events in a computational model are `lapply()` and its closely related variant `sapply()`. Both commands are implemented in R using the same basic syntax:

```
output <- lapply(input_vector, function_to_apply, other_args)
output <- sapply(input_vector, function_to_apply, other_args)
```

in which `output` is the name of the object that will store the results produced once the actions/events have been performed by all units; `input_vector` is an object or vector containing multiple elements which can be selected (e.g., a vector of values, a data frame with multiple columns, a list with multiple elements) and will serve as the input values to the function; `function_to_apply` indicates the function that should be executed simultaneously for all units; and `other_args` names and defines the values of any additional arguments that can be supplied to the desired function. Note that the implementation of both the `lapply()` and `sapply()` commands are essentially identical; in fact, `sapply()` was designed to be a "simplified" (hence the "s" in its name) version of `lapply()` that produces more user-friendly output. The only difference between these two commands is that `lapply()` always

return the output of its operations as an R list whereas the `sapply()` command attempts to return results as a vector, matrix, or an array (as appropriate) if it is able or an R list otherwise. Consequently, the choice between which of these commands to use in model code concerns only how the model-builder wants the data generated during this step to be structured. Because lists in R can be a less intuitive data structure to work with when initially learning to code in R, the remainder of this section will reference only the `sapply()` command. However, the following information is applicable to using `lapply()` as well.

There are two important considerations when using an `sapply()` statement to model simultaneous actions/events. The first concerns the `function_to_apply` portion of the command. The `function_to_apply` can be any function or command that is accessible in the R environment where the `sapply()` command is invoked, including those loaded by default in R (e.g., `mean`, `sample`), imported by an external package, or custom built. It is also possible to build and implement a custom function entirely within an `sapply()` command. Such functions are referred to as anonymous functions in R. The syntactical structure, restrictions, and requirements for constructing and using an anonymous function are like those for creating any custom function in R. However, an anonymous function built entirely inside an apply command is considered local and therefore will not exist outside of the apply statement in which it is used. Consequently, writing an anonymous function in an apply command is most useful when the function is relatively simple (i.e., involves few steps) and/or unlikely to be needed elsewhere in the code.

A second important consideration when using `sapply()` to perform parallel actions/events across units concerns the `input_vector` and its relation to the function supplied to the apply command. Similar to how the `function_to_apply` within an apply statement is referenced, the object used for the `input_vector` may be one that already exists in the R environment when the `sapply()` command is invoked (e.g., an initialized vector called `age` that contains values for all simulated individuals' ages) or can be specified directly in the apply statement (e.g., writing `1:10` as the `input_vector` argument of the `sapply()` command indicates that a vector containing the values 1-10 should be used as the input to the function). In either case though, the elements included in the `input_vector` will *always* and *automatically* be used to define the *first* argument of the function specified in `function_to_apply`. If a function requires and/or allows other arguments to be passed into it, these must be labeled and assigned permissible values after declaring the function as part of the `other_args` arguments in the

apply command. Alternatively, it is possible to control where the values included in the `input_vector` are used in an `sapply()` command by creating an anonymous function, as demonstrated in the forthcoming example.

Modeling simultaneous actions/events for units using an `sapply()` command can be understood as follows. First, the first element from the `input_vector` is selected, assigned to the first argument provided in `function_to_apply`, and the function operations are then performed using that argument definition. After the function executes under this parameterization, the next element from `input_vector` is selected, and the process repeats until all the input elements have been used. Once the function has been run using all the input elements, the results from this entire procedure are returned. Although this operation resembles how a for-loop operates, the `sapply()` command carries out the action/event computations specified in the function for each input element "internally" and thus there is no potential to access, use, or store any of the intermediate results computed for each unit. Consequently, the actions/events modeled in the `function_to_apply` can only be carried out independently from one another and therefore the specific order in which the modeled actions/events are executed across the elements/units of the `input_vector` cannot be accounted for.

Figure S2 demonstrates how an `sapply()` command could be used in computational model code. This code produces the same result as that shown in Figure 4 of the main manuscript which used a for-loop to implement the custom `learn()` function across a set of simulated individuals. The only differences between the code shown in Figure S2 and Figure 4 are in the Model Algorithm section and, specifically, how the `know_pool` object is updated for each agent.

As noted in the main manuscript, the theory represented by this model did not posit that an individual's attempt to learn a piece of shared information affected how this process operates for any other individual. Consequently, using a vectorized approach rather than a loop to model agent learning in this case is appropriate. Because the `sapply()` command effectively performs the learning action/event for all members simultaneously, it generates the changes to all agents' knowledge pools at the same time. As such, the output of this step is written directly to the `know_pool` object rather than to each row of this object (i.e., `know_pool[i, ]`) as in Figure 4. Similar to the approach used when constructing the for-loop in Figure 4, the vectorized code in Figure S2 uses `1:n_members` as the `input_vector` to the apply statement to indicate which units are learning.

The example in Figure S2 also shows one way in which an anonymous function can be created inside an `sapply()` command to execute a simultaneous action/event. This is accomplished by using the `function()` command used to create custom functions in R as the `function_to_apply` in the apply statement. In this example, the one and only argument to this anonymous function is named `learner` and, as previously described, will automatically adopt the values provided in the `input_vector` of the `sapply()` command (i.e., `1:n_members`). The body of the anonymous function then uses the previously created `learn()` function to enact the learning action/event and supplies it with the appropriate input values similar to its use in Figure 4. In this case, the `learner` parameter is also used to specify the location of each agent's knowledge pool and cognitive ability level from the `know_pool` and `cog` variables (respectively) passed to the `learn()` function. Thus, this particular implementation of the learning action/event using an `sapply()` command closely resembles how this step was accomplished using a for-loop.[3]

Of final note, when a function used by an `sapply()` command returns a vector and every vector produced is the same length after all elements of the `input_vector` have been used, the `sapply()` command will column-bind the resultant vectors to create a matrix. In other words, the output produced when the `function_to_apply` is computed using the first value of `input_vector` will be stored in the first column of the output matrix, the output produced when using the second value of `input_vector` will be stored in the second column, and so on. Thus, care should be taken to ensure that the dimensions of the final object returned when using `sapply()` are in the desired orientation. In the present example, this behavior results in each agent's knowledge pool being stored as a column in the resultant `know_pool` object produced by the apply statement, which is reversed with respect to how this object was initialized. The final line of code in the Model Algorithm section in Figure S2 thus uses the transpose function `t()` to produce the desired orientation for the object.

How do I Run Simulations with a Model?

A notable value of building computational models to translate theory into code is the capacity to execute the formalizations, operations, and ideas developed within an explanatory account on a computer and generate data based on with the assumptions/specifications of that theory. Indeed, virtually all computational models are constructed with the intention of conducting *model simulations* in which a model's parameter values are systematically varied to create unique initial conditions, introduce alternative action/event sequences, or allow for

different operationalizations of proposed mechanisms (Vancouver & Weinhardt, 2012). To this end, model simulations can be conducted for many purposes. There are numerous resources describing potential uses of, principles for validating and interpreting, and recommendations for conducting and reporting model simulations (e.g., Ballard et al., 2021; Davis et al., 2007; Farrell & Lewandowsky, 2010; Harrison et al., 2007; Kozlowski et al., 2013; Kozlowski et al., 2016; Naylor & Finger, 1967; Rand & Rust, 2011; Sargent, 2013). However, most of these resources assume one already has the requisite knowledge to develop and run a simulation study with a computational model and therefore do not discuss the process of writing code to conduct model simulations to generate data. To provide one such example and offer a point of reference for future learning, we outline the steps involved in writing code to perform simulations in R. Specifically, we demonstrate one method for writing R code to generate data from the completed version of the example shared knowledge development in teams model provided in the "shared_knowledge_model_basic" model script.[4] To keep this model script separated from the simulation code, we provide an additional downloadable R script entitled "shared_knowledge_model_simulation" that contains all the code necessary to run a simple simulation study with this model.

Before discussing the process of writing model simulation code, it is important to recognize that a simulation's purposes often shapes the needs, construction, and organization of model code (i.e., what model parameters are included, which parameters are fixed versus variable, what data objects are needed or useful, which and how actions/events are controlled, etc.). Consequently, writing model code that represents the logic of a computational process theory and writing code for purposes of conducting simulations with that model can be seen as separate yet interrelated activities. We thus advise focusing first and most intently on writing model code that accurately represents one's proposed explanatory account before attending to any additional considerations that running model simulations might entail. Note also that we do not discuss considerations when designing simulation studies, making choices about which and why to manipulate specific model parameters, wrangling simulated data, conducting analyses/visualizations with simulated data, or interpreting/validating computational models using simulated data. These are obviously important activities when conducting and reporting model simulations and further guidance on these topics can be found in the previously cited references, but they extend beyond the scope of our present discussion.

There are multiple approaches that could be used to write code capable of simulating a computational model to generate data in R. The general procedure for running model simulations we recommend capitalizes on the design philosophy of the R programming language and involves three main steps: (1) adapting a completed computational model script into a function; (2) identifying and preparing the data objects to be saved from a single model run; and (3) writing simulation code that instructs the computer to automatically run the computational model using different parameter values. We briefly summarize the changes introduced into the example code provided in the "shared_knowledge_model_simulation" script to help make the focal activities of these tasks more concrete; however, many of the fundamental concepts necessary to implement these changes were covered in the main manuscript (e.g., using functions, running repeated computations using loops) or these supplemental materials (e.g., code vectorization, data structures). That said, we will also introduce a small number of additional programming commands and concepts in the present demonstration that help to improve the efficiency with which a simulation study is constructed and executed in R.

Creating a Computational Model Function

The first step in our recommended approach for writing code to simulate a model script involves adapting the *entire* body of code written for the computational model into a single function in R. Although this may sound like a complicated task, it tends to be very straightforward and begins by simply placing the entirety of the finalized code for a model inside the body of a new function:

```
model_function <- function(parameter_args) {
      # Model parameters code
      # Model functions code
      # Initialization code
      # Model algorithm code
      # Data output code
}
```

In this sense, the goal is to create a new function in R which contains the instructions for carrying out *all* the operations necessary to enact the steps of the computational model when it is called.

The next key coding activity in this step entails identifying what parameters will be manipulated as part of the simulation, adding those parameters as arguments in the new model function, and making the necessary adjustments within the computational model code to accept those arguments. In effect, this activity entails specifying the *inputs* of the computational model function that will be passed into the model code and used to create the conditions under which a given model run will be performed. The choice as to which input variables to specify at this stage will be directly informed by the intended goals/purpose of the simulation study as these are what dictate which aspects of the model (i.e., parameters, functions, etc.) should be manipulated. In the provided example, we demonstrate how to adapt our computational model function so that it can be used to manipulate the average extraversion, conscientiousness, and cognitive ability levels of simulated team members. This change allows one to run a simulation study with the model in which the goal is to explore, say, how team composition across these attributes would be expected to affect the acquisition of knowledge within and across team members given the assumptions of the proposed explanatory account. Consequently, the procedure used to initialize these three attributes in the model code need to be adapted to allow values for these parameters to be easily changed for different model runs.

To accomplish this goal, we start by examining the procedure used to initialize simulated team members' level of extraversion, conscientiousness, and cognitive ability. In the "shared_knowledge_model_basic" script, these values are initialized by randomly sampling from a uniform distribution with given minimum and maximum values. In the "shared_knowledge_model_simulation", we revised these operations to create members such that average team-level value for each of these parameters could be manipulated between model runs (see lines 101-124):

```
ext <- rep(ext_mean, n_members) + rnorm(n_members, 0, sd = .05)
cons <- rep(cons_mean, n_members) + rnorm(n_members, 0, sd = .05)
cog <- rep(cog_mean, n_members) + rnorm(n_members, 0, sd = .05)
```

This procedure assigns the same value to all agents on a simulated team for a given parameter and then adds a small amount of "noise" around this value for each member to reflect between-individual variability. The result of this procedure is that all members will, on average across model runs, have extraversion, conscientiousness, and cognitive ability levels equal to `ext_mean`, `cons_mean`, and `cog_mean`. Because the operationalization for the `ext`, `cons`, and `cog` parameters in the current model specification are restricted to the [0,1] range, the revised

initialization code also includes operations to adjust any values sampled outside of this allowable range to permissible values. The final change necessary to implement this change is to include the newly created `ext_mean`, `cons_mean`, and `cog_mean` variables used in this initialization procedure in the argument list for the new computational model function (see line 2). This allows us to assign values to each of these variables that are automatically passed into the model code used to generate attributes for simulated individuals when the computational model function is called.

Preparing Output Data from a Model Run

Whereas the previous step focused on creating the model function and preparing the desired input parameters for a simulation, the focus of this second step concerns preparing the desired outputs of the model function and the data that should be saved from each model run. As before, decisions about what data should be saved from a model run will ultimately be informed by the goals and intended purpose of the simulation study. For purposes of the present example, we will record two data objects generated by the computational model that are likely of conceptual interest: how many pieces of information each simulated individual knows at each time point (i.e., the `agent_know` matrix data object) and the speaking record (i.e., the `spoke` matrix data object). Additionally, we will also save the initialized attributes of the simulated individuals generated for each model run (i.e., the `ext`, `cons`, and `cog` objects) so that an exact record of each simulated team's composition is recorded.

In the previous step, the entire computational model code was converted into a function. As noted in our prior discussion on working with model functions in R, all the data and data objects generated inside an R function are considered local to that function and thus cannot be accessed once it has finished executing unless explicitly included as part of a `return()` statement or printed as the last command of a function. We also previously noted that the `return()` command used to determine the output of a function can only execute a single expression/operation; if we wish to return multiple data objects from a function, those objects must be combined into a single object and referenced in the return statement of the function. Taken together then, the primary coding task of this step is to create a single data object containing all the data that we wish to save from the model run and reference that object in a `return()` command included at the end of our model function. As a reminder, it is important to ensure *all* the desired data one wishes to save from a model run and analyze for a simulation study is

accounted for during this step because it will not be possible to access any of the data or data objects generated by the computational model after the model function is executed.

Line 351 in the "shared_knowledge_model_simulation" code demonstrates how we may complete this step for the present example. Note that because we want to save multiple data objects of different types and organizations in a single data object, we elected to combine these objects into an R list inside the `return()` statement. The resultant output of this model function will thus be a list object containing three named elements. The first list element will be a two-dimensional data frame (`agent_attributes`) containing the extraversion, conscientiousness, and cognitive ability levels initialized for each agent. The second list element (`agent_knowledge`) will contain the `agent_know` matrix data object which records how many pieces of information each agent knows at each time point. The final list element (`information_sharing`) contains the `spoke` matrix data object which records which agent spoke (column 1) and what piece of information piece they shared (column 2) at each time point.

Running a Simulation Study

Completing the prior steps produces an executable function that allows the values of specified model inputs/parameters to be set and model data to be generated using a single command in R. For example, selecting and running lines 1-358 in the "shared_knowledge_model_simulation" script creates a new function in the R environment called `team_knowledge_model()` that contains three arguments (`ext_mean`, `cons_mean`, and `cog_mean`). Assigning (permissible) values to these arguments and then executing the function (e.g., `team_knowledge_model(ext_mean = .3, cons_mean = .5, cog_mean = .75)`) will execute all the code written to run the computational model and return data consistent with the previously described organizational structure. The final step needed to conduct a simulation study with a model thus involves writing code to automate the process of assigning desired values to the model parameters included in the model function, executing the function using those parameter settings, and recording the generated data from each model run.

Our recommended approach for carrying out these procedures to run a simulation study in R is demonstrated on and can be executed by running lines 366-446 of the "shared_knowledge_model_simulation" script. To improve computational efficiency and speed up the amount of time needed to run a simulation study, the simulation code we provide relies on an external package called *snowfall* (Knaus, 2022) that allows R to perform computations in parallel across multiple CPUs on a computer. By default, R only performs computations on a single

CPU. However, all modern computers possess more than one CPU, and allowing R to split up and execute model runs across multiple CPUs can greatly reduce the time necessary to run a simulation study. The *snowfall* package simplifies this process, and thus several of the operations included in our simulation code (e.g., `sfInit`(), `sfExportAll`(), `sfStop()`) refer to commands provided through this package that prepare and regulate R for running parallel processes. Generally speaking, no modifications should be necessary to these calls when adapting the provided code to run a different simulation study.

The two most significant coding tasks for preparing to run a simulate study involve creating the parameter values to use in a simulation study and running the simulation itself (see lines 390 and 433, respectively, in the example simulation script). We prefer to approach the former task by constructing a simulation design table. A *simulation design table* resembles an experimental design table commonly used in empirical research to summarize the factor levels assigned to different study conditions. For purposes of model simulation though, we prefer to organize simulation design tables as two-dimensional data tables in which each column corresponds to a parameter of interest in the simulation study (i.e., the model inputs that will be manipulated in the simulation), each row corresponds to a single model run, and each cell in the table stores the values that will be assigned to a given parameter for a given model run (see Figure S3). In this respect, each row of the simulation design table stores a single combination of parameter settings that will be used for a single run of a model.

The primary coding task involved in creating a simulation design table thus entails writing a series of operations to generate the desired parameter values to be used for every model run in a simulation study and recording them in a data table that adheres to the previously described model run by parameter structure. The procedures used to systematically generate parameter values for a simulation study typically involves either randomly sampling values for parameters from a specific probability distribution (i.e., Monte Carlo sampling) or systematically “sweeping” through a specified range of values for each parameter to ensure that every possible combination of the desired parameter values is included in the simulation study. The provided simulation code implements the latter approach by using the `expand.grid()` command in R to create a simulation design table containing all possible combination of the values between 0.1 and 0.9 in increments of 0.1 for the conscientiousness and cognitive ability parameters we wish to manipulate and pass as inputs to our computational model. For purposes

of this demonstration, we did not manipulate values for the extraversion parameter and thus set this value to a constant (1) for all model runs.[5]

The utility of constructing a simulation design table in the form described here is that it allows one to write code to select a model run (i.e. row) from the design table, extract the values for each model parameter to be used for that model run, pass those values into the appropriate arguments of a computational model function, execute the model under those settings, and then iterate this process until every parameter configuration included in the design table has been selected and all the desired model runs have been completed. Executing this procedure in code essentially involves "looping" through each parameter combination stored in the simulation design table and running the model using the corresponding settings. In the example simulation code we provide, we take advantage of the computational efficiencies gained through vectorized operations and parallel computing by using the `sfClusterApplyLB()` command from the *snowfall* package to carry out this procedure rather than using loops. Extensive comments on how to use this command to run a simulation with a simulation design table and computational model function are provided in the example code.

As with running all functions in R, it is important to ensure that the operation used to execute the simulation study is assigned to a named variable so that its data output can be saved and accessed. In the "shared_knowledge_model_simulation" script, the results of the simulation study are saved to an object named `sim_dat`. The structure of this data object is an R list in which each element contains the data generated by a single model run under the corresponding parameter settings. That is, the first element of this list data object will contain the data generated by the computational model using the parameter values found in the first row of the simulation design table, the second element will contain the data generated using the parameter values found in the second row of the simulation design table, and so on.

Of final note, the procedure described here requires that the computational model function used to generate data with a simulation must be accessible from the R environment in which the simulation script is executed. This is most easily accomplished by placing the code used to initialize the computational model function before the code used to run the simulation study in the same R script file and running the entire code base (as is done in the provided "shared_knowledge_model_simulation" file). However, it is possible to save the computational model function and simulation code as separate files by using R's `source()` command to ensure that the model function is

initialized/accessible from the simulation code. Readers interested in using this approach are encouraged to consult the R help manual for more information on how to use the `source()` command to externally execute code and save the results into the currently active R environment.

Footnotes

[1] The present manuscript focuses on building models using only the functionality provided in base R. However, packages are emerging within the R ecosystem that attempt to provide more direct capabilities for developing computational models and running simulations. For example, the *simecol* (Petzoldt & Rinke, 2007) and *villager* (Thelen et al., 2022) packages offer architectures and helper functions for writing and simulating system dynamics and agent-based models, respectively, within R. Alternatively, the *RNetLogo* package (Thiele, 2014) provides an interface for operating, executing, and communicating with models programmed in NetLogo through R that allows one to use the programming language/philosophy of the former and the analytic tools/capabilities of the latter from the same software application.

[2] It may be worth noting that the base family of apply commands in R do not enact truly vectorized computations in that they do not "split-and-perform" operations simultaneously across separate computer processing units. Instead, R's apply commands are wrappers to functions that internally execute for-loops either directly in R (e.g., `apply()`) or in the underlying C language in which R is written (e.g., `lapply()`, `sapply()`). This implementation has the potential to produce faster and more efficient computations, but many have noted that this behavior is effectively "loop-hiding" rather than vectorization (e.g., Burns, 2011). This distinction, however, is largely irrelevant for the purposes of representing computational process theory and building computational models. Because the internal execution of loops carried out by apply commands in R does not provide access to the "memory" needed to track interdependent effects unfolding across units, they cannot represent any consequences of the specific order in which actions/events are executed (i.e., local interdependence). With respect to building computational models then, R's apply functions can be treated as performing vectorized operations that more efficiently execute and evaluate the results of actions/events simultaneously across all the units on which they operate.

[3] Contrary to our prior recommendations for constructing model functions, the anonymous function created inside the apply statement of Figure S1 does not include a `return()` command to indicate what it should return. Nevertheless, running the apply statement using the anonymous function still produces the desired output (i.e., the updated knowledge pools of all agents). This behavior occurs because if a return statement is not explicitly included

as part of a function, R will return the result of the last expression evaluated inside a function by default. In the present case, this result is the output returned by the `learn()` function, which is the desired behavior.

[4] The same basic process we outline here can be followed to construct simulation code for running the "shared_knowledge_model_intermediate.R" model script as well.

[5] Note that many computational models (including the example model) involve stochastic elements such that running a model more than once using the same parameter values may produce different results. Simulation studies are therefore often designed such that each combination of possible parameter settings used to run a model are simulated numerous times. To replicate model runs using the same parameter settings with the approach outlined here, additional rows containing the duplicated parameter settings should be included in the simulation design table. The code used to construct the simulation design table provided in the "shared_knowledge_model_simulation" script demonstrates one way this could be accomplished. Specifically, that code allows one to specify how many teams should be simulated (i.e., how many model runs should be performed) for each parameter combination by providing a value to the `n_teams` variable on line 381. This value is then used to control how many times each parameter combination appears in the design table using the `apply()` and `rep()` commands in R. For example, we (arbitrarily) set the default value for the `n_teams` variable to 250 in the provided code. This means that each parameter combination will appear 250 times in the simulation design table and thus result in a simulation with 9 * 9 * 250 = 20250 model runs.

References

Ballard, T., Palada, H., Griffin, M., & Neal, A. (2021). An integrated approach to testing dynamic, multilevel theory: Using computational models to connect theory, model, and data. *Organizational Research Methods, 24*, 251-284.

Braun, M.T., Kuljanin, G., & DeShon, R.P. (2018). Special considerations for the acquisition and wrangling of big data. *Organizational Research Methods, 21*, 633–659.

Burns, P. (2011). *The R Inferno*. Retrieved from https://www.burns-stat.com/pages/Tutor/R_inferno.pdf.

Chambers, J.M. (2000). Users, programmers, and statistical software. *Journal of Computational and Graphical Statistics, 9*, 404-422.

Davis, J.P., Eisenhardt, K.M., & Bingham, C.B. (2007). Developing theory through simulation methods. *Academy of Management Review, 32*, 480-499.

Farrell, S., & Lewandowsky, S. (2010). Computational models as aids to better reasoning in psychology. *Current Directions in Psychological Science*, *19*: 329–335.

Fioretti, G. (2012). Agent-based simulation models in organization science. *Organizational Research Methods, 16*, 227-242.

Harrison, J.R., Lin, Z., Carroll, G.R., & Carley, K.M. (2007). Simulation modeling in organizational and management research. *Academy of Management Review, 32*, 1229-1245.

Kozlowski, S.W.J., Chao, G.T., Grand, J.A., Braun, M.T., & Kuljanin, G. (2013). Advancing multilevel research design: Capturing the dynamics of emergence. *Organizational Research Methods, 16*, 581-615.

Kozlowski, S.W.J., Chao, G.T., Grand, J.A., Braun, M.T., & Kuljanin, G. (2016). Capturing the multilevel dynamics of emergence: Computational modeling, simulation, and virtual experimentation. *Organizational Psychology Review, 6*, 3-33.

Knaus, J. (2022). *snowfall: Easier cluster computing (based on 'snow')*. https://CRAN.R-project.org/package=snowfall.

Macal, C.M., & North, M.J. (2010). Tutorial on agent-based modeling and simulation. *Journal of Simulation, 4*, 151-162.

Naylor, T. H., & Finger, J. M. (1967). Verification of computer simulation models. *Management Science, 14,* 92-103.

Petzoldt, T., & Rinke, K. (2007). simecol: An object-oriented framework for ecological modeling in R. *Journal of Statistical Software, 22*(9), 1-31.

Pickle, B. (2022, December 13). *Data*. TechTerms. Retrieved from https://techterms.com/definition/data.

Rand, W., & Rust, R.T. (2011). Agent-based modeling in marketing: Guidelines for rigor. *International Journal of Research in Marketing, 28*, 181-193.

Sargent, R. G. (2013). Verification and validation of simulation models. *Journal of Simulation, 7,* 12-24.

Thelen, T., Aldana, G., Thomson, M., & Gonzalez, T. (2022). *villager: A Framework for designing and running agent-based models* [R package, version 1.1.2.]. https://github.com/zizroc/villager.

Thiele, J.C. (2014). R marries NetLogo: Introduction to the RNetLogo package. *Journal of Statistical Software, 58*(2), 1-41.

Vancouver, J.B., & Weinhardt, J.M. (2012). Modeling the mind and milieu: Computational modeling for micro-level organizational researchers. *Organizational Research Methods, 15*, 602-623.

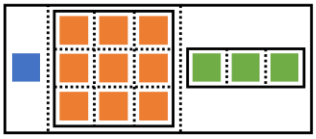
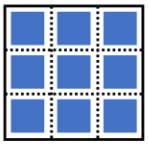
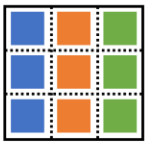
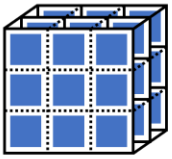
Venables, W.N., Smith, D.M., & the R Core Team (2022). *An Introduction to R*. Retrieved from https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.

Wickham, H. (n.d.). *Advanced R* (1st edition). Retrieved from http://adv-r.had.co.nz/.

Wickham, H. (2019). *Advanced R* (2nd edition). Retrieved from https://adv-r.hadley.nz/index.html.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data science: Tidy, transform, visualize, and model data* (2nd edition). Retrieved from https://r4ds.hadley.nz/.

Figure S1. The five fundamental data objects and structures in R

	Homogenous data	Heterogenous data
1-dimensional	*Vector* 	*List* 
2-dimensional	*Matrix* 	*Data frame* 
Multi-dimensional	*Array* 	

Note. Each cell of the figure presents a notional visualization of the data objects permitted by the five basic data structures available in R. The solid black boundary lines for each visualization are intended to signify a complete data object (e.g., an entire vector, an entire data frame). The dashed boundary lines within a visualization are intended to mark off locations where data can be stored within a data object (i.e., elements of the data object). The colored blocks within a visualization are intended to represent a singular piece of data stored in a data object (i.e., a number, Boolean value, character string). Different block colors within a visualization are intended to represent different types of data stored in a data object (e.g., logical data, numeric data, string data). Adapted from Wickham, H. (n.d.). *Advanced R* (1st edition). Retrieved from http://adv-r.had.co.nz/.

Figure S2. Example of using a vectorized approach in computational model code in R

```
# Model Parameters
n_members <- 5
n_info <- 30

# Model Functions
learn <- function(info, lrnr_know, lrnr_cog) {
  lrnr_know[info] <- lrnr_know[info] + lrnr_cog
  lrnr_know[lrnr_know > 1] <- 1
  return(lrnr_know)
}

# Initialization
know_pool <- matrix(0, nrow = n_members, ncol = n_info)
cog <- runif(n = n_members, min = .1, max = .33)

# Model Algorithm
info_shared <- sample(1:n_info, size = 1)

know_pool <- sapply(1:n_members, function(learner) {
       learn(info = info_shared,
             lrnr_know = know_pool[learner, ],
             lrnr_cog = cog[learner])
       })

know_pool <- t(know_pool)

# Data output
info_shared
```

```
[1] 24
```

```
know_pool
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
[1,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
[2,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
[3,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
[4,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
[5,]    0    0    0    0    0    0    0    0    0     0     0     0     0     0
     [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23]     [,24] [,25]
[1,]     0     0     0     0     0     0     0     0     0 0.2769015     0
[2,]     0     0     0     0     0     0     0     0     0 0.2468860     0
[3,]     0     0     0     0     0     0     0     0     0 0.1733489     0
[4,]     0     0     0     0     0     0     0     0     0 0.2991037     0
[5,]     0     0     0     0     0     0     0     0     0 0.1400874     0
     [,26] [,27] [,28] [,29] [,30]
[1,]     0     0     0     0     0
[2,]     0     0     0     0     0
[3,]     0     0     0     0     0
[4,]     0     0     0     0     0
[5,]     0     0     0     0     0
```

Figure S3. Example structure of a simulation design table with three parameters

Model Run	Parameter 1 Value	Parameter 2 Value	Parameter 3 Value
1	0.3	0.3	0.3
2	0.4	0.3	0.3
3	0.5	0.3	0.3
…	…	…	…
8	0.3	0.4	0.3
9	0.4	0.4	0.3
10	0.5	0.4	0.3
…	…	…	…
50	0.3	0.3	0.4
51	0.4	0.3	0.4
52	0.5	0.3	0.4
	…	…	…
341	0.7	0.9	0.9
342	0.8	0.9	0.9
343	0.9	0.9	0.9

Note. In this example, each of the parameters is permitted to take on one of seven values ranging from 0.3 to 0.9 by increments of 0.1. The R code to reproduce the exact values provided in this table is `expand.grid(seq(from = .3, to = .9, by = .1), seq(from = .3, to = .9, by = .1), seq(from = .3, to = .9, by = .1))`.