A PROGRAMMER BEHAVIOR BY TASK

The previous statistics in Figure 5 were aggregated across all participants (and hence tasks). We now investigate differences across the tasks the participants solved. Table 4 shows the acceptance rate of suggestion by task as well as the top 3 CUPS state by time spent. We first notice that there is variability in the acceptance rate; for example, the difference between the acceptance rate for the 'Data Manipulation' and 'Classes and Boilerplate Code' tasks is 17.1%. When we look at the most frequented CUPS states for participants in these two tasks, we notice stark differences: those in the data manipulation task spent 20.63% of their time thinking about new code to write and 16.48% looking up documentation online, while those in the boilerplate code task spent most of their time verifying suggestions and prompt crafting (=56.36%). This could be due to the fact the boilerplate code is very suitable for an AI assistant like Copilot while the data manipulation requires careful transformation of a dataset. However, we find that 'Verifying Suggestion' is in the top 3 states in terms of time spent in the coding session for all but two tasks, indicating similar behavior across tasks.

Table 4: Acceptance rate and the top three CUPS states in terms of time spent as a fraction of session time for each of the tasks. We include standard errors of the acceptance rate aggregated across participants.

Task Name	# Suggestions	Acceptance Rate %	Top 3 States (time %)
Algorithmic Problem	124	30.6 ± 26.6	Verifying Suggestion (25.58) Writing New Functionality (22.31), Thinking About New Code To Write (19.23)
Data Manipulation	238	24.8 ± 22.6	Thinking About New Code To Write (20.63) Looking up Documentation (16.48), Prompt Crafting (16.38)
Data Analysis	114	29.8 ± 32.3	Debugging/Testing Code (21.23) Editing Last Suggestion (16.62) Prompt Crafting (16.00)
Machine Learning	162	33.9 ± 23.7	Looking up Documentation (19.98) Verifying Suggestion (19.01) Debugging/Testing Code (12.52)
Classes and Boilerplate Code	112	41.9 ± 34.9	Verifying Suggestion (30.34) Prompt Crafting (26.02) Writing New Functionality (13.56)
Writing Tests	83	55.4 ± 49.7	Verifying Suggestion (20.79) Debugging/Testing Code (19.68) Writing New Functionality (16.91)
Editing Code	117	23.9 ± 24.6	Verifying Suggestion (30.18) Editing Last Suggestion (14.65) Writing New Functionality (14.24)
Logistic Regression	74	55.4 ± 35.1	Verifying Suggestion (30.28) Editing Last Suggestion (25.60) Writing New Functionality (15.69)

B PREDICTING CUPS FROM TELEMETRY

Objective. To scale some of our insights, we need to be able to identify and predict programmers' CUPS state. We discuss how we can use telemetry data to predict using machine learning classifiers the CUPS state of the programer. This would enable us to accomplish two goals: 1) use the predictive models on the fly to perform interventions in the user interface and 2) use the predictive models to label previously collected telemetry with CUPS states to perform retrospective analysis such as in section 6.

Setup. The telemetry dataset represented as $\mathbf{D} = \{D_i\}$ collected in our study contains, for each user i a list of events occurring in the corresponding as D_i. An event is defined as a segment of the telemetry that culminates in a shown, accept, or reject programmer action (refer to Figure 2). For the purpose of this analysis, we only retain the shown events (labeled as "User Typing or Paused" in Figure 3) [8]. The list of

[8]Note that consecutive shown followed by either accept/reject events share the same suggestion and prompts and so are very difficult to distinguish from only telemetry.

events for programmer i is $D_i = \{x_{ij}, y_{ij}\}$ where x_{ij} is the features for the event j and y_{ij} is the CUPS state for the event j. Our machine learning models will aim to predict the label y_{ij}. We extract features x_{ij} for each event as follows: the length of the document, previous actions, suggestion features (e.g., suggestion length), the confidence reported by Copilot, presence of Python keywords (e.g., `import`, `def` `try`, etc.), and the output of the Tree-sitter Parser [9]. Finally, we extract features of the prompt including its textual features and parser outputs. It is crucial to note that the model features do not leak any information about the future and can be computed as soon as a suggestion is generated by Copilot.

Experimental Results. Using a leave-one-out programmer evaluation strategy where we train on data of 20 programmers and leave out one programmer for testing, we train an eXtreme Gradient Boosting (XGB) [9] model for this task for each trial (21 total) and evaluate the accuracy on the test set. The XGB model achieves an average accuracy of 30.8 % ± 1.9. In comparison, a baseline that always predicts the majority state achieves 24.9% ± 3.0 accuracy, indicating that the XGB model has non-trivial performance – through there is considerable room for improvement. Nevertheless, while the accuracy reported is low, if we restrict the task to just predicting the most common state of Thinking/Verifying Suggestion (the rest is background) we obtain an area under the receiver operating characteristic curve (AUC) of 0.69 ± 0.02 which shows good predictive power. This shows that there are signals in the telemetry to be able to predict CUPS states. However, this XGB model accuracy is not sufficient to power our proposed interventions but perhaps a larger amount of labeled data can help build more reliable models to execute our proposed interventions. We discuss in the future section other avenues to improve the prediction of CUPS states from telemetry.

C DETAILS USER STUDY

C.1 Interfaces

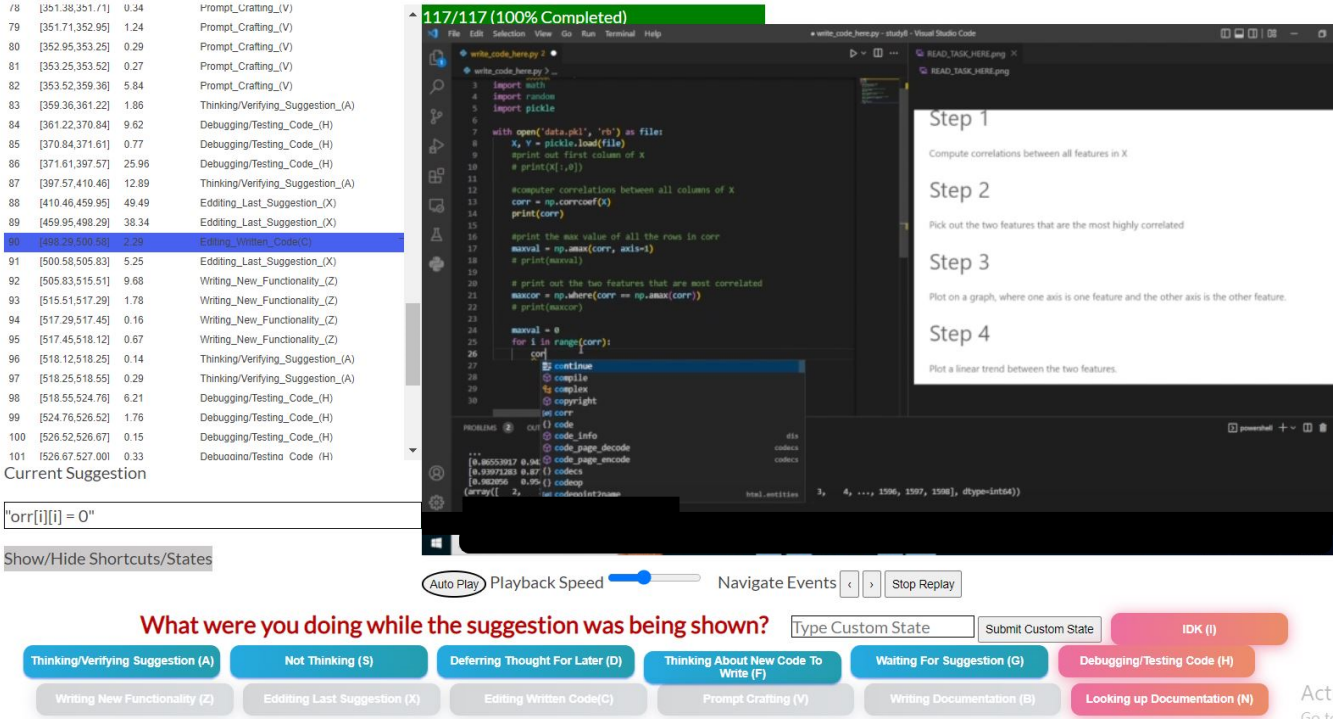

Figure 9: Screenshot of Labeling Tool represented in Figure 4

[9] https://tree-sitter.github.io/tree-sitter/

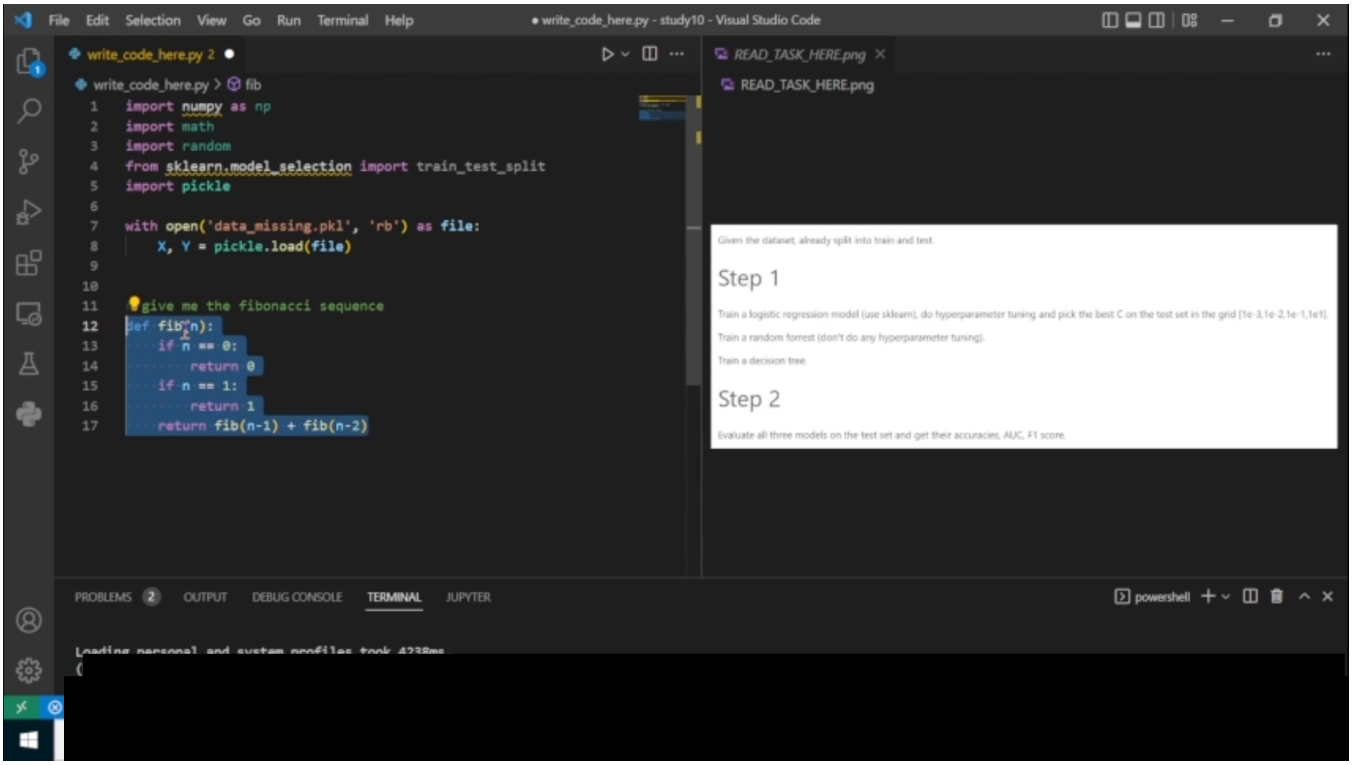

Figure 10: Screenshot of Virtual Machine interface with VS Code

C.2 Task Instructions

The tasks are shown to participants as image files to deter copying of the instructions as a prompt.

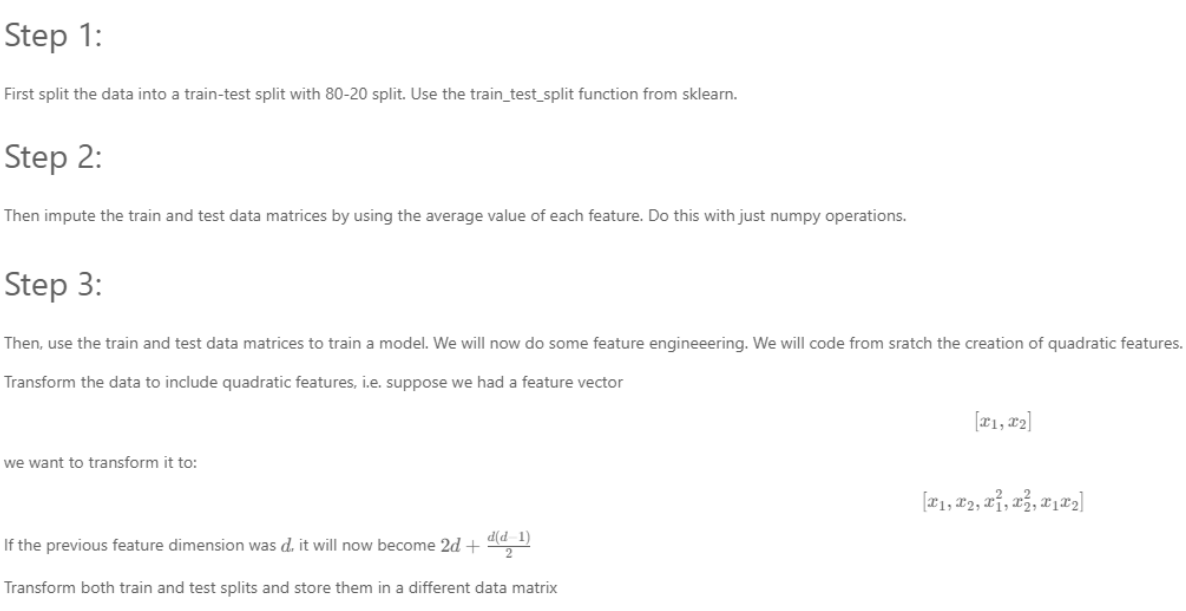

Figure 11: Data Manipulation Task.

Step 1

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9 Output: [0,1] Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Step 2

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

Step 3

Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that:

0 <= a, b, c, d < n a, b, c, and d are distinct. nums[a] + nums[b] + nums[c] + nums[d] == target You may return the answer in any order.

Example 1:

Input: nums = [1,0,-1,0,-2,2], target = 0 Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

Figure 12: Algorithmic Problem Task.

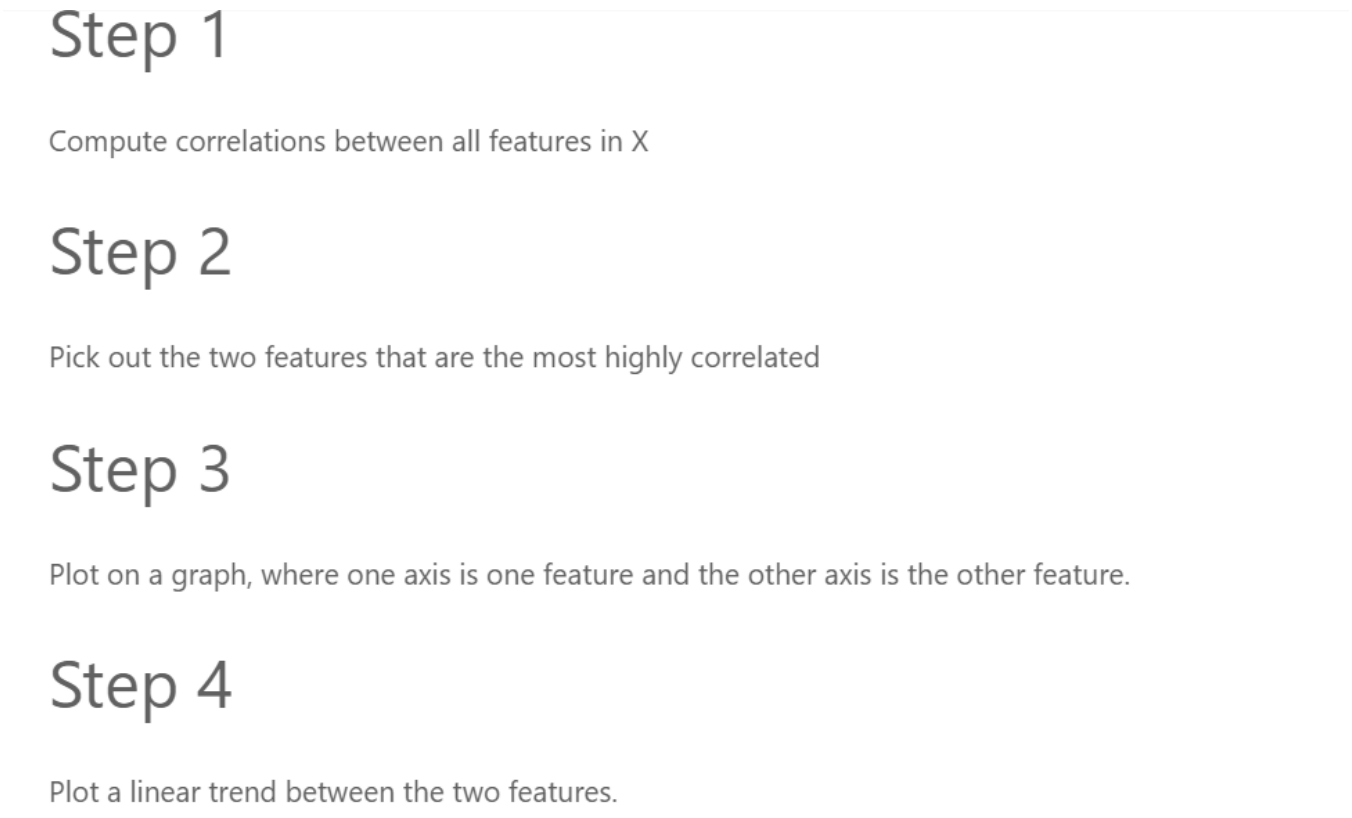

Figure 13: Data Analysis Task.

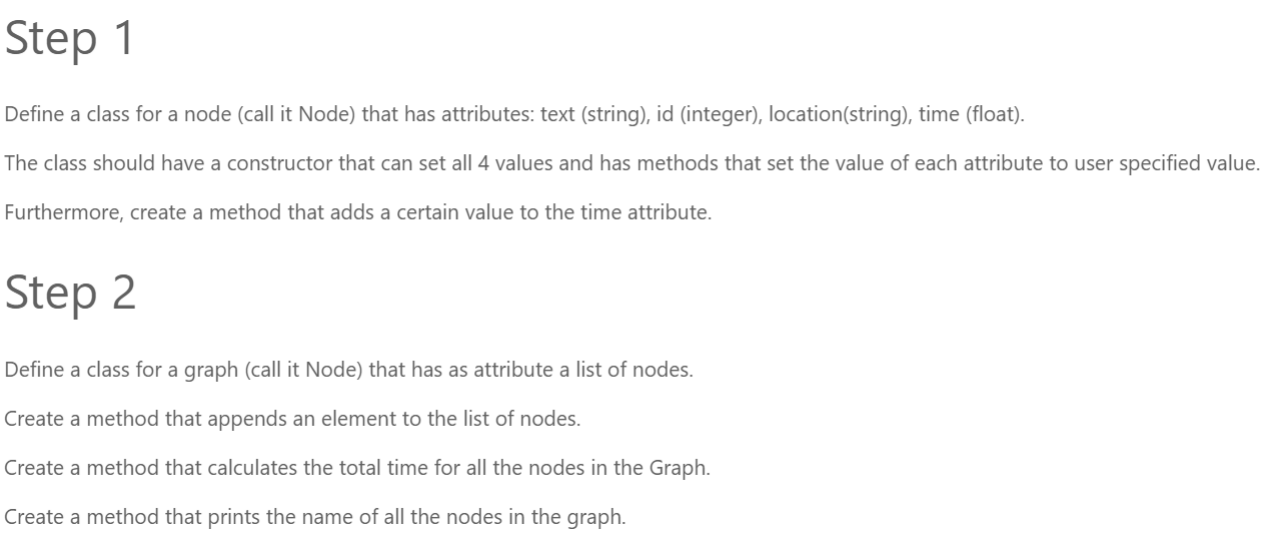

Figure 14: Classes and Boilerplate Code Task.

Logistic Regression

We will implement a custom logistic regression classifier with l2 regularization for this task.

Recall: logsitic regression we learn a weight vector $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$, and predict the probability of the label being 1 as $\frac{1}{1+\exp(-(w^x+b))}$ where this is the sigmoid functio applied to $w^x + b$

To learn the weights, we use gradient descent:

for each iteration we do the update:

$$w \leftarrow w + \alpha * (\sum_i x_i * (Y_i - sigmoid(w_i^x + b)) - 2\lambda w)$$

and

$$b \leftarrow b + \alpha * (\sum_i (Y_i - sigmoid(w_i^x + b)))$$

Implement a logistic regression that can handle custom number of iterations, specified learning rate alpha, specified regularization parameter lambda.

Fit the model on the training data.

Compare the accuracies on the test sets.

Try for 100 iterations, 0.1 learning rate and 1e-5 for lambda.

Figure 15: Logistic Regression Task

Editing Existing Code

Given the following class, this class is a Retreiver which given a set of numerical vectors and a paramter k, can return the k-nearest neighbors of a given vector.

Perform the following edits to the code:

- write a method that returns the least similar k vectors
- write a method that given a set of query vectors, returns the top k vectors for each of the query vectors
- create a method to append new vectors to the already vectors in Retreiver
- create a new distance function that instead of norm we make it a weighted distance as follows:

  Compute maximum scale of each feature on the training set:

  $$scales = [\max_i(X_{1,i}), \cdots, \max_i(X_{d,i}),]$$

  Then let the distance function be:

  $$dist(x, z) = \sum_i \frac{1}{scales[i]} * (x_i - z_i)^2$$

- create a method to change k to user specified value

Figure 16: Editing Code Task

Machine Learning

Training and evaluating a model

Given the dataset, already split into train and test.

Step 1

Train a logistic regression model (use sklearn), do hyperparameter tuning and pick the best C on the test set in the grid [1e-3,1e-2,1e-1,1e1].

Train a random forrest (don't do any hyperparameter tuning).

Train a decision tree.

Step 2

Evaluate all three models on the test set and get their accuracies, AUC, F1 score.

Figure 17: Machine Learning Task

Task

We want to test an api for the following task:

- Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

  An input string is valid if:

  Open brackets must be closed by the same type of brackets. Open brackets must be closed in the correct order.

  Example 1:

  Input: s = "()" Output: true Example 2:

  Input: s = "()[]{}" Output: true Example 3:

  Input: s = "(]" Output: false

  Constraints: 1 <= s.length <= 104 s consists of parentheses only '()[]{}

TODO: Write several test functions to make that the API function isValid(str) works properly.

- Create a class called Testing, inside that class write different test functions that test different aspects of the API (e.g. does it work with '()')), aim for 4 tests.
- Write a method that runs all the tests and returns the average success rate, the standard deviation of the sucess rate.

Figure 18: Writing Tests Task

C.3 Survey Questions Results

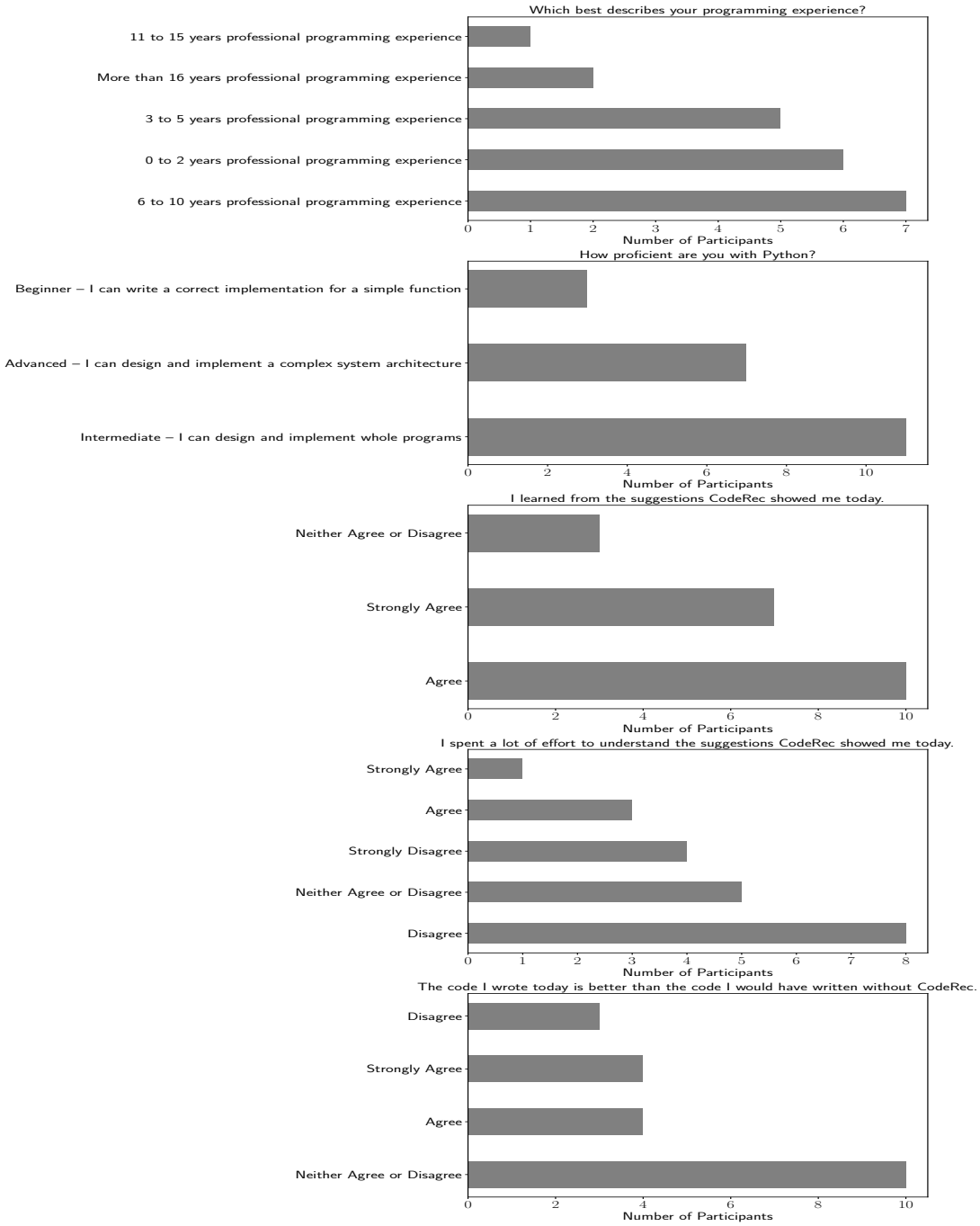

Figure 19: User Study Survey results (1)

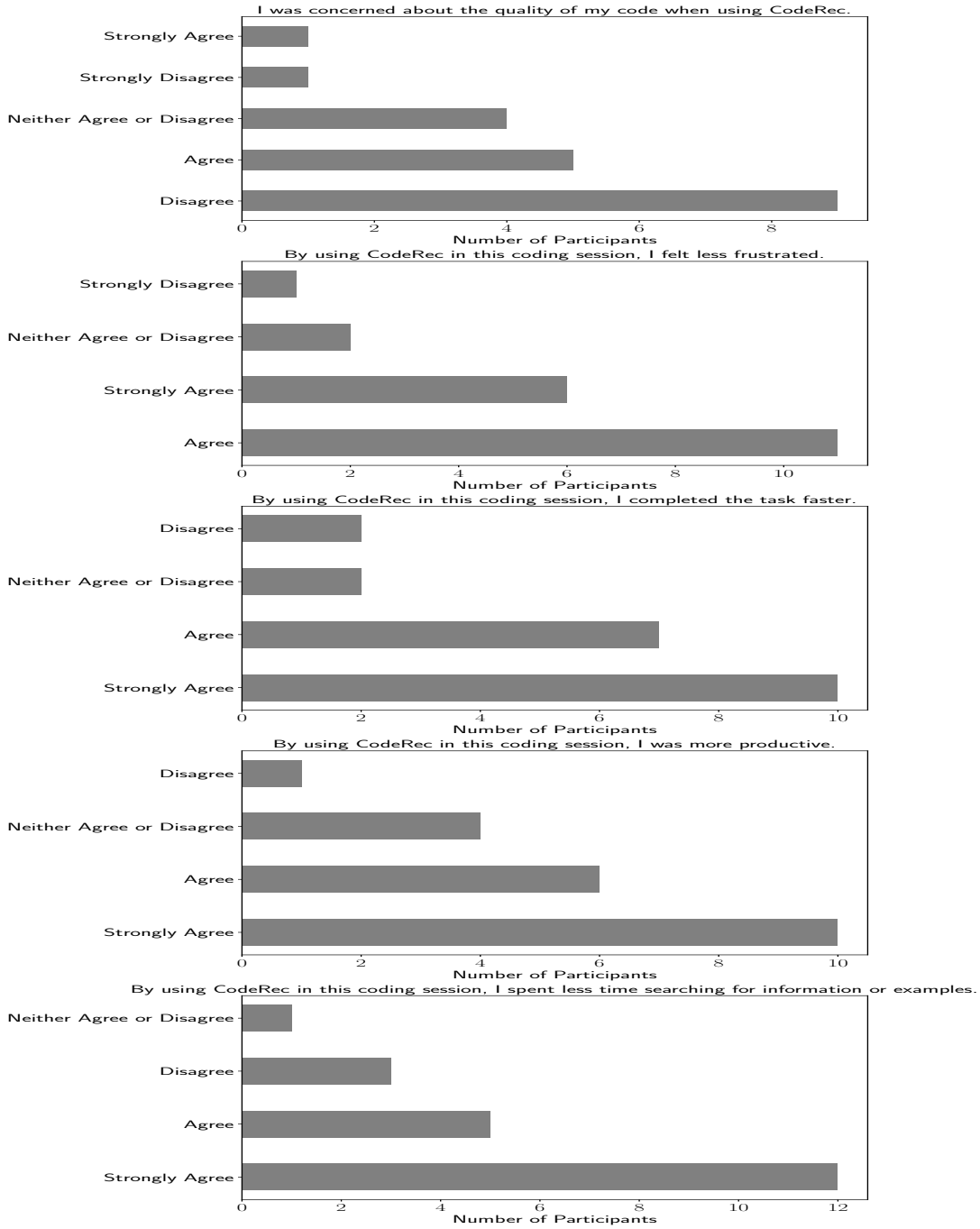

Figure 20: User Study Survey results (2)

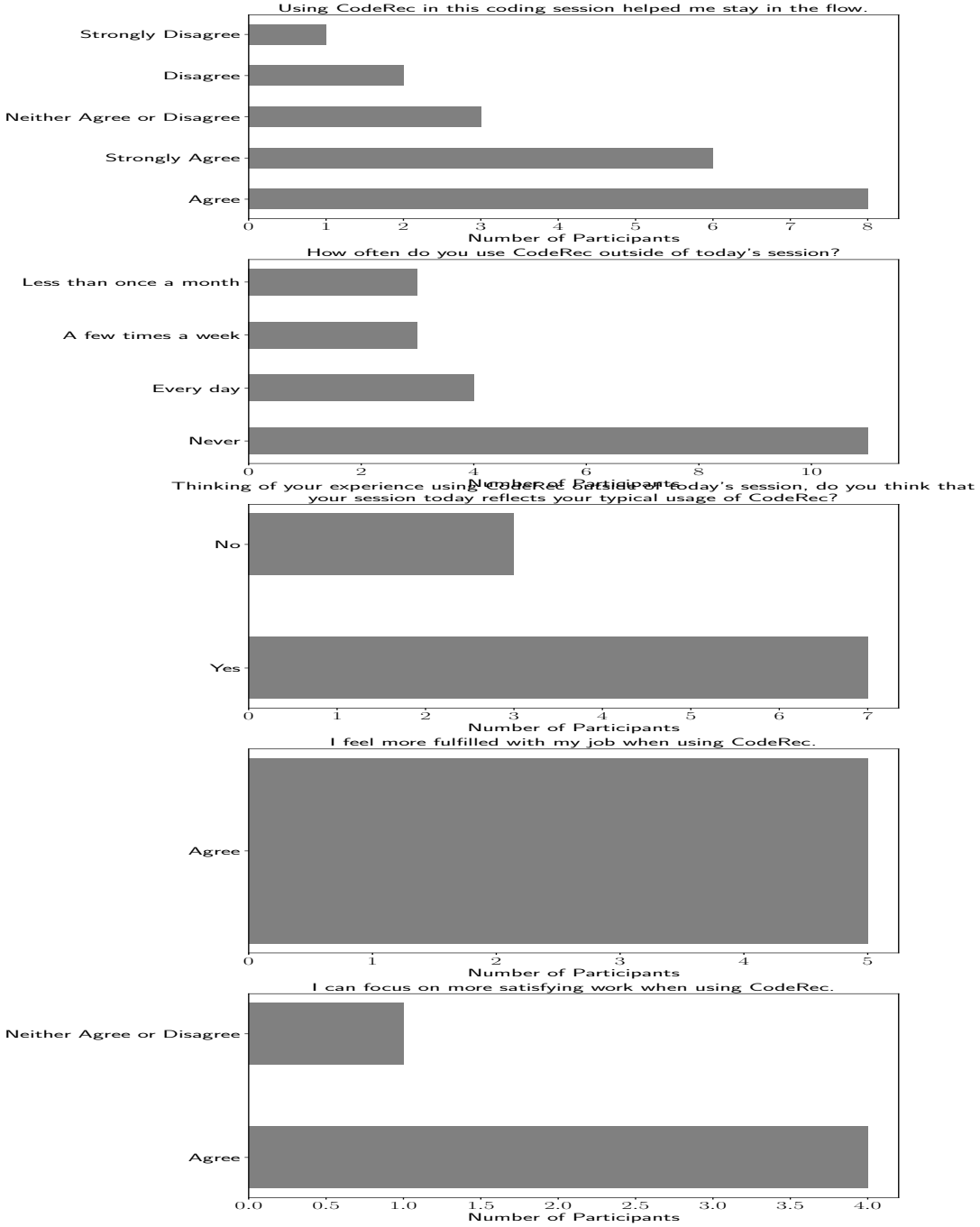

Figure 21: User Study Survey results (3)

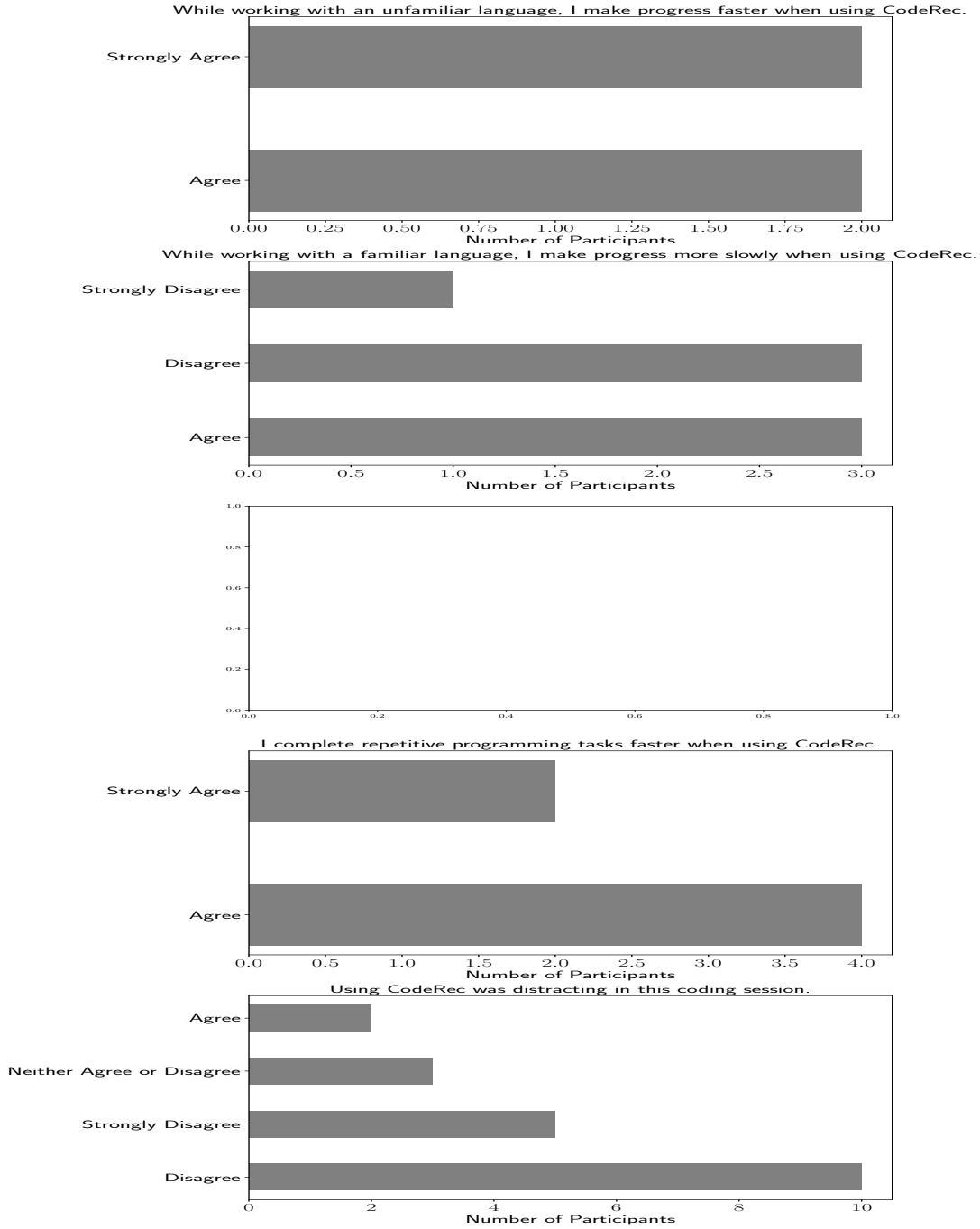

Figure 22: User Study Survey results (4)

C.4 Full User Timelines

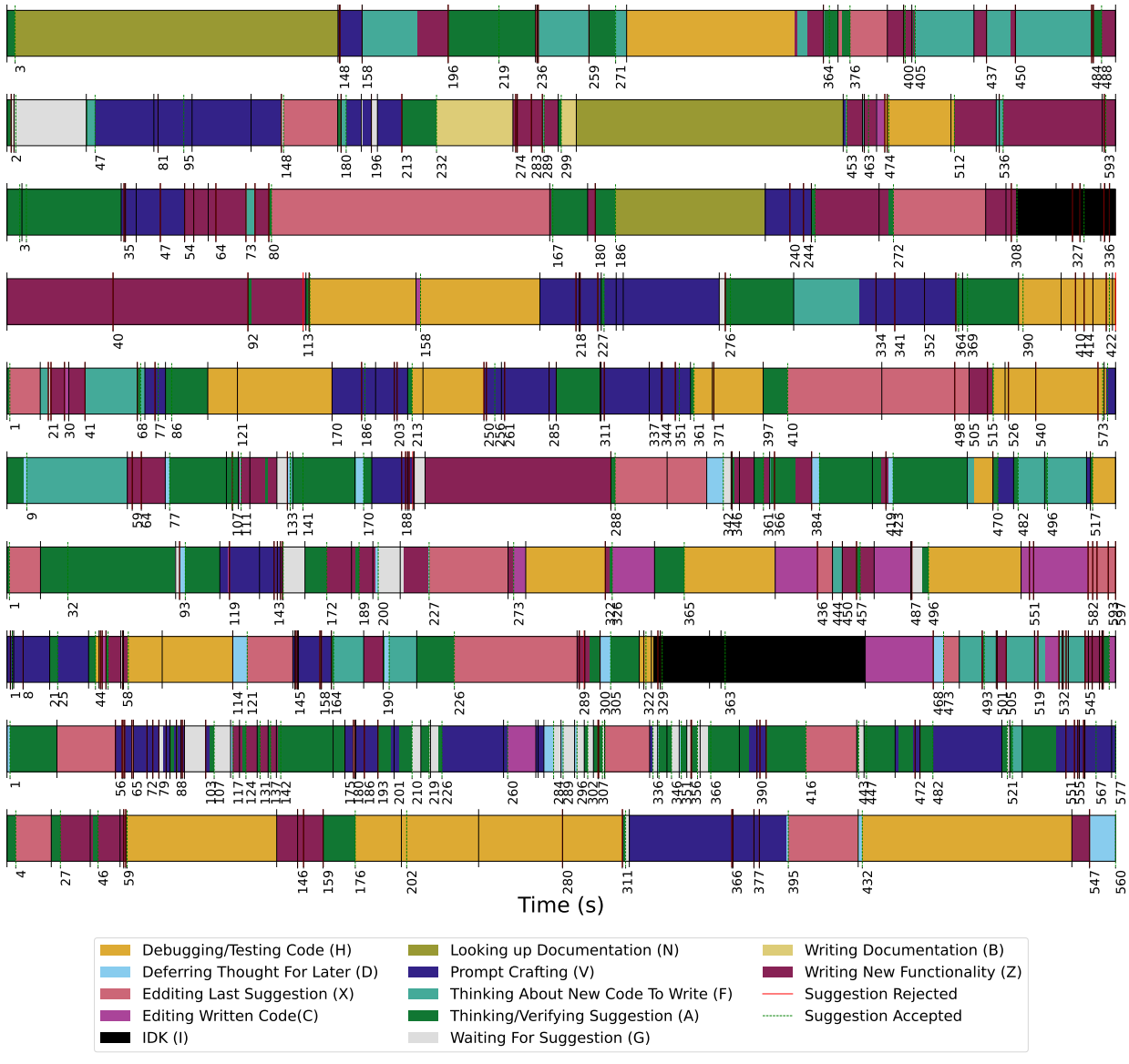

Figure 23: Participants timelines for the first 10 minutes of their sessions (P1 to P10)

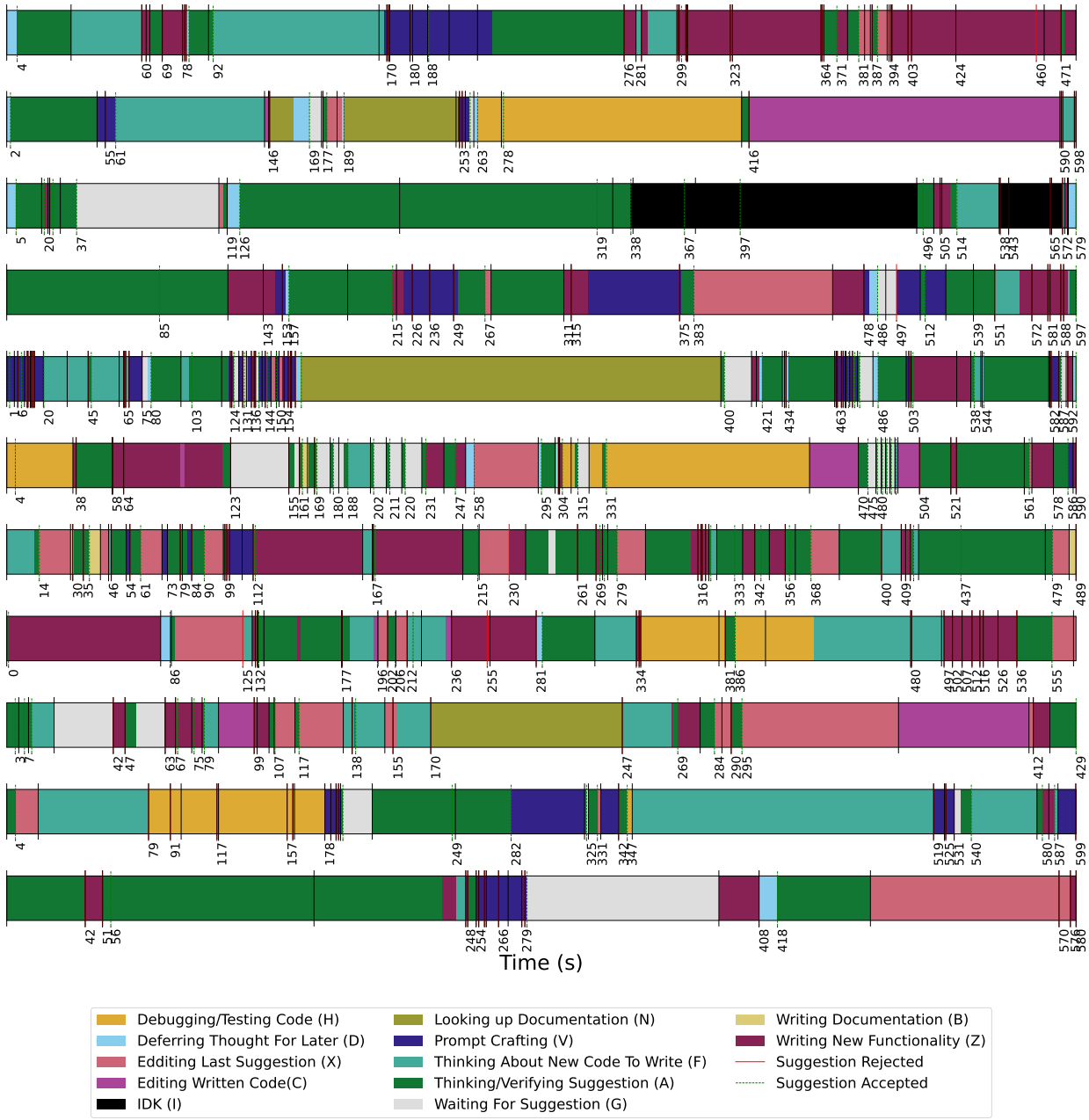

Figure 24: Participants timelines for the first 10 minutes of their sessions (P11 to P21)

C.5 Full CUPS Graph

Figure 25: CUPS diagram with all transitions shown that occur with probability higher than 0.05