Often, when hoping to make even a small change to libraries or similar large software projects, it can be difficult to gain a robust understanding of the architecture, and how bits of code are tied together. Documentation can help with this problem, but many projects (particularly research driven projects) can lack significant documentation to work from. Even for small changes, it can be hard to find what piece of code is the correct one to modify (this is further complicated at times because of in-progress refactoring, and dead code). Once the correct spot is located, though, the lack of familiarity with the project, and overall understanding of its architecture can make it difficult to understand the (potential) side effects of the change, or non-obvious adjustments/call that need to be made with the addition. On top of this, there are often corresponding changes that need to be made elsewhere. Although the change can often be as simple as copy/paste in, or copy/paste with a small modification, but finding the locations where this change is necessary can be difficult, and even require attempting to compile or run the code, and seeing what errors result.

There are a number of tools that exist to attempt to avoid this problem. First, robust, up-to-date, and accurate documentation can allow developers to quickly understand bits and pieces of the code, when reading through them. This can include references to why things are done, which can assist in understanding how to make your change. However, maintaining this documentation, and keeping it up to date is oftentimes onerous, and different developers maintain different standards. Also, this documentation is often meant primarily for people who already understand the project as a whole, or even as a note for the developer who wrote it, for when they return to that bit of code. In these cases, there can be terminology used, and assumptions made of understanding that prevent casual outside developers from fully understanding this documentation. Higher-level documentation, including code maps, and architecture diagrams can help with understanding how system link together. These, however, are also hard to create and maintain, and quickly get out of date. Automatic tooling that can generate both styles of documentation can help, potentially, by attempting to understand the code, and its architecture, and either annotating it, or allowing someone to inspect via the tool (build into an IDE, for example). While these can miss bits of the ‘why’, these tools could assist developers new to a project to understand overall structure, and pinpoint where to make small changes. This could both assist people newly joining the project, and also people who simply wish to make a small change to contribute.

Another issue commonly found with people new to a software project, is a lack of understanding of the overall coding practices. This can mean everything from whether braces go after the statement or on a newline or either (and this may be different for different statements) to preferences towards more complex one-line operations (list comprehensions, etc.) versus more expanded coding style, as well as how they are presented. This can also include aspects of inline/infile documentation, or even organizational/architectural structure of the project. Some organizations have style-guides, or follow ones from external groups (such as PEP), and these even extend down to team level, at times. However, these are not always consistently followed, and oftentimes a team-level one may be lacking, but a standard may be followed. Also commonly the main standard will be followed, but a few exceptions are made with the

team, but these are not documented anywhere. Much of the standards must be learned from team members, or more commonly, simply through working and gaining familiarity with the code. Documenting these can be onerous, and leading to the documents not existing, or quickly becoming out of date and obsolete. Also, given that different languages often have different standards, both inherent to the language, and because of general practice, the number of documents and things that would need to be documented can be barriers to this documentation existing/being accurate. This seems like a place where an automated system could monitor/analyze code, and using some stats/machine learning could build a rough understanding of certain coding standard choices (particularly easy cases might be binary (/trinary) options such as brace positions, etc.)