RISKS: ID5

This document details the potential risks and issues that could arise during the creation of QM-Lab™, as well as some possible solutions. **Please add a risk name in bold all caps, with a description of the risk, as well as a possible solution if applicable.**

RISK NAME: This is a description of the risk. It lists tools/technologies/people etc that the risk concerns. It includes a description of what would happen if the risk occurred, explains why that would be bad, and gives some vague estimate of how likely it is to happen.

- Finally it should describe a possible solution, if possible. Some risks may not have an obvious solution.

- Risks that are likely and dangerous are highlighted in red.
- Risks that are unlikely but quite dangerous are highlighted in green.
- Depreciated risks are at the end of the document in a grey font.

TEAM LEAD DROP: We have some members of the team that are leads for certain important portions of the development work that we do. One of the team leads could drop or change his/her mind about being that lead, which would leave us without someone to manage and facilitate that particular area. This would be bad because that person probably had detailed knowledge of the progress of the team and what its plans were.

- An obvious solution is to have one of team members take over as lead for that team.

UPDATE(ID3): This actually happened and we acted according to the procedure.

DEPENDENCE ON REALTIME API: We are super dependent upon the realtime api at this point. It seems like a little sideproject of a small team at Google. What if they decide it isn't worth continuing to maintain? We could be scrambling to make an alternative work if Google just decides to shutdown the api. Also since it's a free thing they have literally no obligation to keep it open, provide customer service, etc… Seems fairly unlikely to happen in the next 3 months, but possible down the line.

- A partial solution would be to design the software to be as independant as possible. This could be done by using generic methods for updating, creating, etc in the data model. Then it would be easier to replace just the code in those standard methods if we would have to transition to node or some other platform.

HACKERS: Other teams in the past have been compromised, what if we also become compromised? This could destroy our entire project. It could make us look like fools. Maybe osgood would take pity on us and estimate a high grade that we wouldn't have actually gotten?

- The solution: Do not get hacked. Keep back-ups. Be extra secure.

API KNOWLEDGE ISSUES: Only a handful of people are knowledgeable about the API's were using neither API has much documentation that we know about and can easily find.

- Trial and error, use a bunch of functions until we find one that works. Spread the knowledge. Don't be afraid to ask.

COMMUNICATION ISSUES: The team leads may not communicate often enough with each other, and with the teams. This may result in redundant functionality or lack of task completion. For example the design team could create things that the implementation team can't create, because of lack of communication.

- Frequent meetings, scheduled communication, and religious use of communication channels such as Slack and emails.

LACK OF FOCUS: Lack of structure in meetings, we don't often have much of an agenda. Even if we have one, we may not stick to it. Also we may spend a lot of time just hanging out talking.

- Have a meeting lead, that keeps everyone on track. Use agenda's and plan the meeting topics, and stick to it.

TASK FIXATION: We may spend too much time on trivial things, and just kind of lose track of the big picture, we can get caught down in the weeds. Perfectionism can be an issue.

- Triage by the triage team, or the team leads. Better tracking of the big picture. Structured assignments for people and delegation. People take up a specific feature and really keep track of responsibilities as well as time spent on tasks.

SCHEDULING CONFLICTS: We each have different class schedules, as well as our own responsibilities and things we need to do in our own personal schedules. It's unlikely that the entire team will all happen to have a free spot in their schedule that everyone can meet during.

- Smaller meetings with less people, usage of communication channels like Slack, Skype and email. Clear coordination of meeting times so people can adjust their schedule accordingly.

TEAM MEMBER CHANGES: Members may switch between different parts of the team. Every time someone switches from one team to another, they take all of that knowledge and expertise with them, as well as require being caught up by the new team. Requires a replacement on the old team potentially. Since they're new to the other team, they may do things in a different manner, slowing the team down.

- It's not like the old team can't just ask about things they're wondering about. If someone switches into a new team, be more watchful of their work and make sure to help them get up to speed properly so they're not out of the loop.

TEAM CONFLICT: Members could become stressed and resentment could build up between members or even entire teams (testing vs implementation). Everyone is on a full schedule, and possibly underslept so fights and conflict could arise. Some members may get attached to ideas and this could cause arguments.

- Everyone needs to continually remember to behave professionally and be diplomatic. Frequent communication between members may help resolve misunderstandings. Conflict resolution through project manager, and possibly through the instructor.

BUILD UP OF OTHER RESPONSIBILITIES: For example the internship program, lots of people needed to create cover letters and send in applications and do interviews. Other classes have assignments, projects, and tests that require our attention as well. The perfect storm of responsibilities can build up all at the same time, making it very difficult for the team to work effectively.

- Work harder, and more efficient time allocation. Assign earlier deadlines than the officially supplied ones. Good communication of members responsibilities, so that other members can take over other responsibilities for them.

OUTDATED DOCUMENTATION: Documentation that is old and does not have current information can lead to members being misinformed about the current status of our project. This is related to communication issues, because documentations' main role is to communicate information about the project to members so that everyone is on the same page. For example someone could write code in NodeJS not realize we have switched to Realtime API JointJS.

- Update documents frequently, at least for each deliverable.

MEETING ABSENCES: Having a lack of members attending meetings leads to further loss of communication, members get left behind and are unable to work on the project because they are unsure of what needs to get done or the way in which it should be completed. It also places a heavier workload on the members that are present because there are less helping hands to complete necessary tasks.

- Our solutions include: Effective communication of meeting times, time tracking to help accountability, surveys to help accountability, meeting minutes to get people caught up.

MEMBER DROPS : We have a risk for team lead drops, but not one for member drops. This is a risk that has actually happened. The fallout has been that we lost a member on the testing team, which lead to a higher strain on the remaining members of that team. There was also a loss of knowledge and familiarity with the current testing efforts. He had detailed documents and spreadsheets for test matrices, and he is also hosting our Google drive folder with our documents.

- Our solution has been to add a new member to the testing team to relieve some of the workload and offer assistance in helping with the work that needs to get done to thoroughly test the project. He is simply going to leave the documents on his Google drive where we can freely access them.

MEETINGS VS WORK SESSIONS: The vast majority of our meetings have been too focused on decided what to do, rather than simply allocating time to get work done. It is important to communicate and have plans, but it's also very important to actually spend time working on tasks related to the project. Working a group is very effective in comparison to working solo, and having work sessions can help us get more work done, and have superior communication to trying to work solo. Having too many meetings and too little work sessions may have caused people to attend less frequently because the meetings can be uninteresting at times, while work sessions tend to be more enjoyable and productive.

- Our solution is to attempt to have more work oriented meetings where the goal is to work as a team on the tasks related to the project. The way to do this is related to our strategy to have more meetings in general, which means having accountability so that members attend, keeping an eye on scheduling conflicts, and having clear communication of meeting times so that people know when and where they are taking place.

UNSOLVABLE ISSUES : At times we have bugs or problems that we need to fix, and if a certain bug or issue was unfixable, either because it's too difficult or there's not enough time. For example a bug could arise that is the result of a core implementation choice, or the overall structure of the project that would be too time consuming to fix. This would be bad because it would be a broken feature or bug that is in the final project, which the stakeholder would not like. We could waste time trying to fix it, when we could have spent that time fixing other issues. It could be a security hole, which has its own risks which include getting files stolen or modified, malware being served up by us, the project being destroyed and other downsides.

- Depending on what kind of bug it is, it may be possible to educate users to only use the program in a certain fashion. Feature removal may be required to remove the bug, we could also make certain operations illegal that make that bug arise. This requires knowledge of how to reproduce the bug. Another way to prevent this risk from happening is to have spike prototypes of new technologies and ideas to make sure they're practical and achievable. We also need to be thorough in how we plan the overall structure, having a consistent coding style and proper modularity will assist in reducing coupling which can help isolate bugs.

GIT FLOW: We have a team git flow that we're using so if someone doesn't follow it, for example by pushing to master instead of development, that will make our git repo more disorganized, and that is an issue because our git repo becomes inconsistent. Things can go missing if people start branching off the wrong branches and merging incorrectly, files can end up where they don't belong. This would result in merge conflicts that take excess time to resolve.

- Documenting our git flow throughly, which Shane created on our wiki. Our group has a communication platform called Slack which has channel that is integrated with git and logs all actions performed on the repository. This allows group members to easily identify issues with other group members git commits, merges and branch creations among other things and help them to correct those mistakes.

GIT INCOMPETENCE: Git is a powerful tool, and that means it has the capability to do a lot of damage. When used incorrectly, it is possible for harmless mistakes to happen that have no significant effect, or for serious consequences that destroy the project codebase and require a large time investment to remedy. This is clearly something we would like to avoid, because our time is limited, and wrestling with git to fix issues that shouldn't have arisen in the first place would be a waste of valuable time and resources that could have been devoted to actually building the project itself.

- The best way to avoid this risk is by having proper education on how to use git correctly, as well as effective communication of the chosen git usage standards and git flows. The wiki currently contains a description of how we plan to use the git repo, it describes the git standard practices. If people adhere to those practices we should avoid git conflicts and other related problems.

TOOL INCOMPATIBILITIES: We are using a multitude of different tools, APIS, languages, technologies and platforms each with their own quirks and specific usage requirements. These tools all function differently and are used for different things, however they need to work effectively together depending on each other's interfaces. There are a large number of things that could possibly go wrong if the tools suffer incompatibilities, including: bugs, unusability, and being caught off guard later on by finding out a certain link for example doesn't work, or some API feature refuses to cooperate.

- Multiple prototypes, and thorough reading of documentation as well as paying attention to known issues can help to make us aware of possible conflicts between different software tools. Testing and experimentation with tools generally helps us to find issues between tools.

INFREQUENT COMMITS - LARGE COMMITS: Not committing code often enough leads to potentially large commits where large amounts of progress are suddenly added to the code base. It is possible that other members had pulled old versions of the code, and incorrectly thought that the project functioned a certain way, when in reality that had been changed already but they were unaware of it because there was no trail of git commits that update the progress of that branch or feature. Because lots of functionality is added at once, the git commit messages are likely to be very large and this makes it difficult to skim them quickly, meaning that important information can be missed. It is also very easy for the commit message writer to forget what features have been added and code that has been modified, so they may forget to mention it in the commit message.

- Our solution to this risk is to remind the members to adhere to proper git usage and standards, as found in the wiki documentation. Doing more frequent commits, and remembering to list all features added in the commit messages can help ensure the messages are accurate and useful to the other team members.

INFREQUENT COMMITS - LOST PROGRESS: Because lots of functionality is added in each of these large commits, it is possible that the person's computer could die or the progress could otherwise be deleted somehow before the commit is made. This would lead to a loss of progress that needs to be redone, wasting time unnecessarily. Other members might rely on certain base functionality before a feature is actually completed, if the code is not committed until it is fully completed other members are left waiting around for the commit before they can begin building on top of it. This is an issue because it again wastes team productivity.

- Our solution to this risk is to remind the members to adhere to proper git usage and standards, as found in the wiki documentation. Doing more frequent commits, and remembering to list all features added in the commit messages can help ensure that new features are added in a timely manner, so that other members can begin using them and building on top of them.

OVERLY FREQUENT COMMITS: If people commit frequently with git, there will be many small commits of code and small messages associated with them. This means that there are a lot of little messages to read in order to get caught up on what has been added or modified, and people may prefer not to read all of the messages. People can become desensitized by all the commits and place less value on pulling often, because they are used to the commits being small and inconsequential. Doing frequent commits increases the possibility that incomplete functionality is committed before it is really ready. Having incomplete functionality, or poorly tested code is likely to lead to bugs. These bugs can then make their way into other people's code, causing confusion and even more bugs as well.

- Our solution to this risk is to remind the members to adhere to proper git usage and standards, as found in the wiki documentation. Doing less frequent commits, and remembering to ensure all features and functionality are complete and rigorously tested, will ensure that bugs are not as likely to be introduced into the code base.

LAST MINUTE IMPLEMENTATION LEAVING NO TIME FOR TESTING: As has happened in other sessions of CMPT 371, it is possible for the implementation team to get carried away coding new features in, adding to old ones and fixing bugs all the way until near the deadline. This leaves little to no time for the testing team to conduct or create thorough tests that ensure the project is working correctly. As a result, bugs are much more likely to slip into the final product because they’re less likely to be found by tests, and they are less likely to be fixed even if they are found because of the time crunch. Additionally grades would be lost simply based on the fact that not enough testing was conducted and it is part of the project grade.

- Have strict deadlines for the implementation team so that the testing team can get to the code, with the help of the implementation team and create plenty of tests for it.

UNCLEAR INTERFACES: Interfaces that use imprecise terminology, such as subset vs proper subset, or local usages of words such as billion (10^12 in UK but 10^9 in NA) leads to confusion and questions about how to correctly use the method. Operational interfaces, rather than declarative interfaces, can be overly restrictive to the method creator while at the same time not providing useful information to the user of the method.

- User clear, precise declarative interfaces that detail the expected inputs and outputs of the method, the assumptions made, the invariants enforced(if relevant) as well as additional information that makes it clear how the method is meant to be used and what it's error conditions may be.

LACK OF MODULARITY: Code that is closely coupled and interdependent is very difficult to scale and can lead to very tricky bugs that are hard to untangle. It can prevent the creation of unit tests because those methods may depend on much of the rest of the project working as a system, such as requiring authentication(which prevents mocking), before these methods can be tested. Obviously a lack of testing is dangerous because it encourages bugs.

- Write well modularized code, where functionality is decomposed into multiple methods with clear singular purposes and detailed interfaces. Do not reach into objects to grab instance variables, instead use public getters and setters. Have classes that serve a well defined purpose, and group functions based on that intended use. Always remember to write DRY code and take advantage of object oriented principles.

MISSED EQUIVALENCE CLASSES: In testing there are different overarching classes or groups of tests that can be run against the code, but obviously there isn't enough time to use all possible inputs so all equivalence classes of inputs need to be tested. If one is missed, that leaves a large surface area of inputs that can potentially cause bugs that would go uncaught until later in the program when it is not clear what has been the cause of the issue. A different, unrelated functionality can break and if the testing team doesn't realize an entire equivalence class of tests has been missed it may be very tricky to narrow down the cause of some bugs.

- Brainstorm as many possible types of inputs that are possible and be sure to have a test case or two that tests each of them. Be thorough during the creation of tests so as to be more likely to cover all the possible equivalence classes. Lastly, get another set of eyes on the tests, a new person may immediately think of new test possibilities simply because they're fresh to the situation.

SHADOWED METHODS: Subtypes and supertypes may evolve independently of each other, and if the subtype overrides a method from the supertype it may not adhere to the same interface that the supertype does, but yet this object can still be pass around the program to many of the same places that the supertype can. This can lead to bugs and unexpected behaviour when a calling function gets results different from how the supertype method would have behaved.

- Some methods and classes should be made final, and subtypes need to be careful when overriding parent methods.

INACCURATE DEFECT COUNT: Having an accurate sense of the number of defects present within a system can be useful in guiding the testing and debugging efforts because it gives a sense of progress and measure of how much may be left to fix or find. A defect count that is inaccurately low can lead to a false sense of security that causes a less serious search for bugs, and potentially less thorough testing because the team assumes there's not much wrong with the project.

- Always be thorough and rigorous about testing, code creation, and adhere to software engineering principles to ensure bugs are less likely and that the code is more maintainable and extensible. A defect party can be held where to two teams separately find bugs for a few hours and then compare how many they each found that were the same and the unique defects they found that the other team didn't, and then use those numbers to estimate how many defects are present in the system. Regularly doing this can give a better estimate of the defect count.

INACCURATE TIME ESTIMATES: It's easy to be overly optimistic about the amount of work that can be completed in a given time, because we often don't realize how much work goes into building simple features, and we often forget to account for setbacks and delays. Having inaccurate time estimates can lead to a team biting off much more than they can chew, overloading themselves with huge amounts of work and giving themselves impossible deadlines because they underestimated the time needed to complete all of the related tasks. This can lead to burnout and dampened moral, because the team thinks it is performing poorly when in reality it just set its sights too high. Stakeholders are also negatively impacted because they're likely to have made plans based on the given estimates, and they can even lose money and business as a result. They will not be happy about deadlines not being met.

- Don't try to estimate huge features, use agile methods to have a shorter development cycle so that features can be planned out on the scale of days or hours rather than months or years. Break down features into sub tasks and smaller features that are well known and easier to estimate the time needed to complete. Make sure to account for setbacks and delays. Most importantly track how long it actually takes and use this to improve the accuracy of future time estimates.

Depreciated Risks

REALTIME API TOO HARD TO USE: The Google realtime API is our current weapon of choice, we will be using it to enable interaction across clients in real time. It could be difficult to use, which generally means that although we can make progress with it, it may be slow progress, or it could be error prone, or just very picky about what you try to do with it. This would be bad because it would slow us down, take more time to debug, and take more energy that could have been spent elsewhere.

- A possible solution would be to use some other API such as Operational Transform.

UNDO FUNCTIONALITY STORAGE SPACE: If we implement an undo function for the QM_Lab “drag” feature, assuming the feature updates the location in real-time with the mouse movement, the potential storage size of the “undo” stack could get very large, very quickly. Since Google’s Realtime API sets a hard limit of 10MB of collaborative storage per document, any undo functionality we include that involves the “drag” feature could quickly overload the file/document. This would be bad, because Google’s Realtime API stops working when the user gets to this limit.

- Possible solutions include storing our own undo stack that only is updated on mouse-up events, but this too could take up Google’s storage space if we’re not careful.

GOOGLE PRIVACY STORAGE: Certain portions of the Google Realtime API store information using Google’s systems.

- Depending on how important privacy is, and where this information is stored (for example, Google has data storage within Ireland), we may need to disclaim that our information does not meet those requirements, or perhaps this is already included with Google’s account setup.

UNKNOWN RATE LIMITING: While google states that the only limits on the api are a document size of 10MB. However, they might decide to block our api key if we get like a lot of users (1k+) and they start noticing unusually high amounts of bandwidth/servertime going to this obscure api they published a couple of years ago.

- Contingency procedures would be the same as for DEPENDANCE ON REALTIME API.

FIND NEW BETTER TOOL: We could find a new tool that is way more useful and effective than what we’re currently using, that invalidates our current work requiring a restart.

- By this point we probably shouldn’t switch.

JOINTJS HIGH LEVEL : Abstraction provided by joint.js could prevent easy creation of low level functionality, especially with Google Realtime API.

- Theoretically we could implement it ourselves because joint.js is open-source. If we tried really hard we could probably figure out a hack-y way of doing it.

JOINTJS INTERACTION WITH GOOGLE REALTIME API: May be unforeseen complications even though base functionality is proven.

- More spike prototypes and back-up tools.