1.

a. The two major problems concerning a software project is cost and how long it will take. I feel that cost is the most important because a project will get nowhere without a sufficient amount of money and if that is too much for investors there will be no project. Complete functionality is the ideal outcome and the two biggest contributors to that is time and money. Investors want low cost and done as soon as possible but the end product has to be completely functional, but can only be completely functional with money and time.

2.

a. In the Agile method for software development, the five main phases that occur in each iteration are gathering requirements, design, code, and test, which collectively produce a working system. While you might be tempted to do some of these upfront, this approach is generally discouraged in Agile. Trying to lock in requirements or design early is a recipe for disaster, often resulting in building the wrong software because customers frequently change their minds and gain understanding as the project progresses. Therefore, in my opinion, attempting to front-load and eliminate any of these phases from every iteration would actually increase the overall project time and risk of failure by reducing the crucial feedback that stems from development that allow for necessary adjustments towards the customer's actual needs.

3.

a. The Waterfall method follows sequential phases: Requirements Analysis, Design, Coding, Testing, and Maintenance- which differs significantly from Agile, where the equivalent phases (Requirements, Design, Code, Test) are repeated cyclically within short iterations to produce working software. Waterfall inherently includes a distinct, dedicated Maintenance phase after delivery, an explicit concept that Agile typically integrates into ongoing development cycles rather than isolating it as a massive, separate final stage. In my opinion, this distinct Maintenance phase, while necessary to consider, is too rigid in Waterfall, as real-world issues and bug fixes often need immediate attention, not formal hand-off to a separate future phase. However, a situation where an Agile team might benefit from dedicating a focused "maintenance-ish" phase is after a major launch to exclusively

address high-priority production bugs and performance tuning, preventing those issues from derailing the team's planned features for the next iteration.

4.

- What is a user story?
    - A user story is a requirement written from your customer's perspective, describing one thing the software will do for them
- What is blueskying?
    - Blueskying is a brainstorming session with your customer and team where you think big and capture every idea for requirements without initial criticism.
- What are four things that user stories SHOULD do?
    - Four things that user stories SHOULD do are describe one thing the software needs to do for the customer , be written using language the customer understands , be driven by the customer , and be short.
- What are three things that user stories SHOULD NOT do?
    - Three things that user stories SHOULD NOT do are be a long essay , use technical terms unfamiliar to the customer , or mention specific technologies.
- Does the Waterfall method have user stories?
    - The traditional Waterfall approach often uses an upfront, detailed requirements document instead of the flexible, ongoing use of user stories characteristic of agile methodologies.

5.

- All assumptions are bad, and no assumption is a good assumption.
    - There is a gray area. You can assume the user has basic computer operation skills and small things like that. You cannot assume that the user automatically knows complex, complicated features.
- A big user story estimate is a bad user story estimate.
    - I agree because you don't want to have a huge time frame for a single iteration. You instead should break it down into smaller sub sections to increase efficiency and accuracy.

6. Fill in the blanks in the statements below, using the following things [you can use each thing for more than one statement]: Blueskying; Role playing; Observation; User story; Estimate; Planning poker.
    - You can dress me up as a use case for a formal occasion: **User story**
    - The more of me there are, the clearer things become: **User story**
    - I help you capture EVERYTHING: **Blueskying**
    - I help you get more from the customer: **Roleplaying**
    - In court, I'd be admissible as firsthand evidence: **Observation**
    - Some people say I'm arrogant, but really I'm just about confidence: **Estimate**
    - Everyone's involved when it comes to me: **Blueskying**

I agree with them except for blank 3 because observation is very subjective and not fact and does not capture the whole picture if you're limited.

7. A better than best-case estimate is a Utopian day estimate, representing the time a programmer thinks a task will take under absolutely perfect conditions, ignoring all real world distractions and project overhead. Programmers often think in these Utopian days, assuming they can crank through the work without interruption. However, developers who consider the real world know that a Utopian day estimate must be adjusted by velocity to account for overhead and convert it into the real world days actually required to complete the task.

8. The best time to tell your customer you won't be able to meet her delivery schedule is as soon as you know there's a problem. I feel that telling the customer as early as possible is the best time because it gives the customer the maximum amount of time to make a hard choice. The customer can then decide what should be done for that iteration or for the project as a whole. This transparency is part of the overall process that keeps the project on track and helps deliver great software on time and on budget. Yes, this would absolutely be a difficult conversation. No customer wants to hear that their software is going to be late. You could make the conversation less difficult by being honest and providing a clear explanation supported by your project data. You should show them your iteration plan and clearly explain why, with the current resources, the work threatens your ability to deliver what they need by the due date

9. Branching in software configuration management is mostly bad unless absolutely necessary. The core problem is that each branch is a separate code base that must be

maintained, tested, and documented. This duplication of effort introduces hidden costs and makes it difficult to ensure that fixes and new features are correctly applied everywhere. If you view branching as a major decision that shouldn't happen often, you'll be ahead of the game. A scenario that supports this view is a team deciding to create a separate branch for developing features for two different operating systems, Windows and Mac. For every new feature developed, the code would have to be implemented in both branches. When a bug is found, it must be fixed in both branches. This effort doubles the workload, confuses versioning, and makes scaling the team unnecessarily difficult. Branching should only be used when necessary, such as maintaining a released version that needs critical patches while major, potentially breaking, development continues in the main trunk..

10.

a. We have used Microsoft Copilot to assist with planning and execution of our website, and although it helps increase efficiency, it definitely has its shortcomings. For starters, you cannot blindly accept the code as truth. A lot of the code it provides is either incomplete or causes errors. This can be a hurdle to try and debug. However, what Copilot is best at is organizing data which has been so helpful to us for the planning phase that the positives of Copilot outweigh the negative.