AMAZON
SOFTWARE DEVELOPMENT ENGINEER
CODING EXERCISE PREP

Coding - Problem Solving

This competency assesses your ability to take something complex, break it down, identify a solution and translate that solution into working code. These questions will most likely have an ambiguous problem statement. It may be more difficult to understand what the problem is and you will need to ask clarifying questions to understand how you can start to solve the problem. The biggest mistake you can make is to listen to the question and then dive right into coding, so make sure you start with those clarifying questions - this is a collaborative environment with your interviewer and they're there to help guide you. If they see you getting off track, they'll also provide hints to get you back on track. So always take those hints and run with them - they aren't there to trick you.

Since the problem is ambiguous you should understand how much time you have to solve the problem, then split the time into 3 tasks:

1.) Uncovering requirements and considering evolution of the situation
2.) Implementing a first pass of the design and covering all requirements
3.) Enhancing the solution, and when satisfied that the original requirements are met, seeking out more requirements to show how your solution evolves when requirements change.
Practice solving problems without an integrated development environment. During the interview, you'll be asked to showcase your coding skills without an IDE. Re-familiarizing yourself with basics typically handled by a IDE program, like correct syntax (your preferred coding language), so you can focus your brain-power on the interesting parts of a solution rather than trying to recall those utilities.

Show how you deal with ambiguity.

For example, demonstrate that you can:
- Handle intentionally vague requirements
- Change strategies quickly when the approach doesn't work
- Work through open-ended questions
- Be adaptable and quick learning

- Think out loud. The whiteboarding challenges are your opportunity to show off your problem-solving skills. We want to hear how you're transforming the solution! So practice talking out loud as you work through your process and solve the problem.
- Keep it simple. Try not to over-engineer the solution to solve for things it's not supposed to.

Wondering what a good response looks like?

Here we give you an inside look at the key criteria your interviewer is looking for, as well as tips and tricks from the pros on how to best address each criteria in your response.

Correctly solve the problem.

- Consider edge cases and inputs that would cause your code to fail.

Justify decisions by identifying potential tradeoffs with several different solutions; understand why one solution is better than another.

- Get at least one working solution, even if it is not the most optimal one. Efficiency and scalability are important, but it is better to deliver first a working solution and raise it is not optimal, rather than not producing working code. Then you can either iterate over it to improve it, or implement a new approach.
- Explain your approach before coding, and vocalize your thought process as you proceed. This isn't natural for everyone, but it will help your interviewer give you good feedback and hints if you need them.
- When vocalizing your points to the interviewer, write different test inputs, try to visualize, and talk about how you would translate it into code and any caveats you might want to consider in your code.

Define the problem by asking relevant clarifying questions.

- Make sure you understand the problem and functional and not functional implications before you jump into code.
- Test your ideas with examples and callout any assumptions you are making - ask the interviewer if they agree with them as you go.

Consider additional factors (e.g., developer effort, team composition) beyond the basic problem.

- When coding, use descriptive variable names, follow consistent indentation, and use best practices and recommended style for the programming language you have chosen.

Require minimal hints and be able to use the hints to reach solutions to the problem.

- Identify test cases (happy and edge ones) and run them through your code to verify your solution works.
- Don't be afraid to ask for help, but make sure you ask directed questions. For example, saying "I'm stuck" is very general and it can be hard to give advice. If you can describe approaches

you're thinking about but don't believe they'll work (and why) or things you've tried this is better.
- There are no trick questions or trick hints, everything will be useful. When the interviewer provides a hint, write it down so you can use them in your solution.

Data Structures and Algorithms

Your interviewer will ask you to solve a data structure-oriented problem. You will then write code using the appropriate data structure to solve the problem. You'll also be expected to compare various approaches and explain why the one you selected was the right one.

This competency measures your ability to choose the most efficient run-time solutions. For example, different data structures are more performant for reading data while others are more performant for writing. Algorithms should be as fast as possible and, of course, solve the problem correctly. Look out for edge cases for which your algorithm fails to satisfy requirements.

- Consider runtimes for common operations and understand how they use memory.
- Understand data structures you encounter in core libraries (e.g., trees, hash maps, lists, arrays, queues, etc.)
- Understand common algorithms (e.g., traversals), divide and conquer, when to use breadth first vs. depth first recursion.
- Discuss runtimes, theoretical limitations, and basic implementation strategies for different cases of algorithms.

Suggested areas of refresh:
- Binary search tree data.
- Scalability methods—With the architecture, there are many techniques and methods which can be used in order to customize and improve the design of a system. Some of the most widely used are: redundancy, partitioning, caching, indexing, load balancing, and queues.

Criteria and pro tips for addressing them:

Use optimal data structures and algorithms to solve the problem:
- Build a toolbox of what data structures would be useful in common scenarios such as efficient access by key/object/position, maintaining sort order, searching, and finding maximums/minimums. Also think about edge cases where your chosen data structure may not function as intended - for example with hash-based data structures what happens when there is a hash collision, or what a degenerate binary tree looks like and why.
- Be familiar with the features, data structures, and algorithms in your programming language(s) of choice. If you dive deep into these it might introduce you to data structures you may not normally see/use so you can expand your toolbox for solving problems.
- If you're comparing two data structures or algorithms that each have some drawback (very common!), pick one and give a justification for it. Remember to explain the benefit, operations, and time complexity.

Identify potential shortcomings and discuss tradeoffs with different data structures and algorithms.

- Clarify the shortcoming/issues you can expect by using the solution you are planning to implement. If you know a better DataStructure that can address those shortcomings but you are not confident in using it in your solution, make sure to mention and discuss this with your interviewer as well before you start with using the DS you are confident to use.
- Try to identify the pattern for the given problem, but don't overcomplicate it. Could it be solved using recursion, say specifically DFS? Could it fit into the bucket of dynamic programming? Can you achieve a solution using a linear data structure like array instead of a non-linear one like heap?

Justify why the selected data structures and algorithm was used.

- There is always more than one way to solve the problem. If you identify other solutions, explain why you are using the one you are, this shows a good understanding of the Algorithm and the problem.
- Be prepared to talk about your understanding of the data structures you choose and answer questions about how those data structures work under the hood. Do you know multiple ways of resolving collisions in hash maps? Why is heap insertion O (log n)? Answering these questions gives us confidence you understand the data structures and have not just memorized their answers.

Demonstrate a solid grasp of runtime and space complexity tradeoffs even if not perfectly accurate O(n) syntax.

- If the interviewer asks you a real-world problem, first zoom out and think through all relevant possible data structures and algorithms that could be used to solve this problem. Before finalizing a solution, clarify problem constraints and consider multiple ways forward, see which are worth discussing before recommending one and going ahead with it.

Coding - Logical and Maintainable:

This competency measures your ability to write maintainable, readable, reusable, and understandable code. We'll explore how you structure your methods and classes to ensure the logical separation of concerns is clear. We'll explore how you name variables, methods, and classes in a way that future developers with no previous knowledge of the code can understand how it works, evolve the logic, investigate it, and debug it when needed. Test names should describe business and technical requirements, and the test code should be consistent with the test names.

Create simple code (e.g., leverage reuse, properly format, no improper coding constructs).

- Feel free to write out examples of how your code will be used. If it seems overly complex, go back and think if that's how you'd want your coworkers to use their code, and if there are ways to simplify it.
- Time is precious; rather than spending time trying to think of an optimal solution start with a working solution, no matter how simple, and enhance it as you go.
- Think about extendibility, make sure you are able to extend the code when/if requirements change, avoid a single function that does everything.

Create maintainable code (e.g., quickly able to trace impact of changes, clear variable naming conventions).

- Make sure that your method, parameter, and variable names are clear & descriptive (e.g. don't just use "a" or "foo") and try to separate out functionality into discrete methods/functions with clear responsibilities. Remember that we'll likely run/operate software longer than it took to write it, so the clearer things are future maintainers will thank you.
- Your code should be reasonably easy to maintain as the traffic or load increases.

Organize code in a way that is easy to read and understand.

- Use functions and classes, and inheritance to break up your solution into logical components. This improves readability and makes it easier to extend the solution for new requirements.

Create syntactically correct code, or it would be with minor improvements.

- Pick the coding language you are most comfortable with - we're not looking for any language-specific knowledge (unless you've been told otherwise). Ensure that you're comfortable with the idioms and conventions of your chosen language (e.g. error/exception handling, managing resources like file I/O or network connections etc.) and use descriptive variable and method names.
- If you need a method call but forget its exact name or arguments, call that out and ask the interviewer if you can make up a name and arguments.

Create code that works as intended.

- Start small, solving one thing at a time, and iterate over your solution as you ask questions or the interviewer does more follow-ups. If you see clear design patterns or abstractions that can be applied from the very beginning, apply those! It is better to realize code can be refactored to support more requirements in a maintainable way, rather than building a complex solution or investing time applying a design pattern that doesn't solve the requirements.
- Think through test cases (both working and breaking), edge cases, boundary conditions, null/nil/none etc. Try and enumerate these cases before you begin coding so that when you have a solution you feel works, you can use these as validation. Be sure to also confirm test cases with your interviewer.

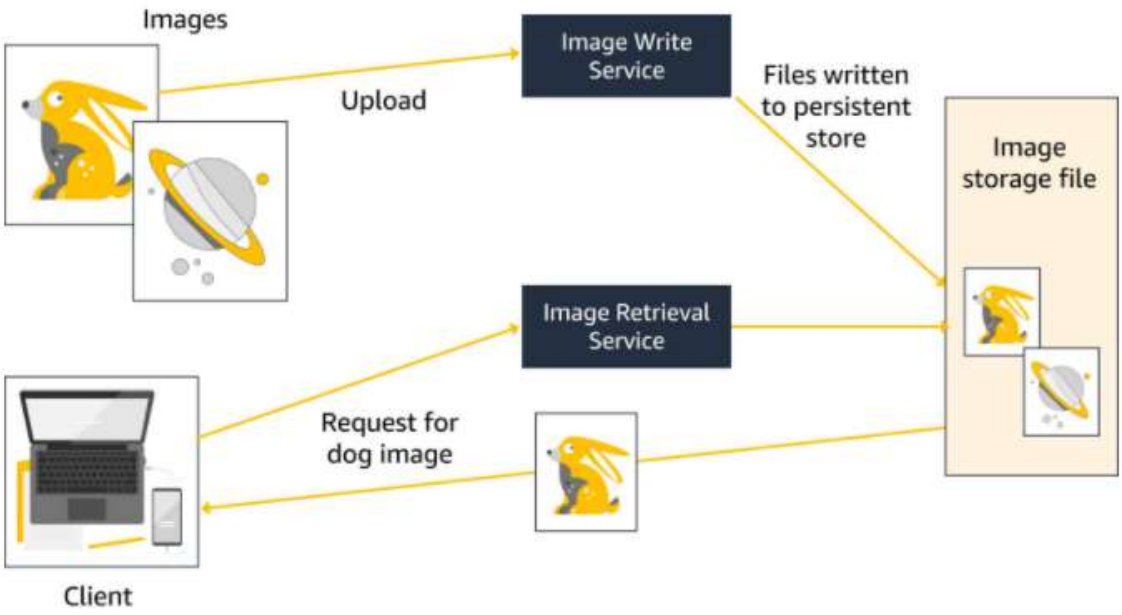
System Design

Good design is paramount to extensible, bug-free, long-lived code. We know it’s possible to solve any given software problem in almost limitless ways, but when software needs to be scalable and maintainable, good software design is critical to success. One way to build lasting software is to use object-oriented design best practices. Think and talk about software components, distributed systems, and system trade-offs.

However, designing these systems at Amazon is special because it happens in one of the most massive, fast-moving, and unpredictable environments of any existing organization. This means that traditional centralized styles will break down far before they can handle the volume of users necessary.

The system design competency is designed to measure your ability to black box design a software system.

The goal is for you to deliver a design in production with considerations of deployment, scaling, failures, availability, and performance. Be prepared to discuss latency and concurrency.



How to best prepare for this competency:

- Often times, software systems need software components, something to store data, something to make decisions (such as business logic) and APIs or processes.
- Knowledge of distributed systems, SOA, and n-tiered software architecture is very important in answering systems design questions. If you don’t work with these concepts regularly, be sure to review them.
- Also, be sure to practice drawing your system design by hand and be prepared to whiteboard.

Note: For the system design competency, there's often more than one correct answer. What matters is your process for how you transform the solution.

Criteria and pro tips for addressing them:

Ask clarifying questions to scope-down and define requirements.

- Your interviewer is there to help you with clarifying questions, assumptions, and providing a customer's perspective. As a designer you should start with the (primary) customer and work backwards:
  - Who are you designing the system for and why?
  - What expectations do they have in terms of functionality?
  - What things would a customer just assume will be in the system but they may not think about in the forefront of their minds? (e.g. it'll be fast and secure)
  - What happens if we become hyper-popular with customers? What does 2x growth look like? Or 10x? And how would that influence the design?

- Understand first what problem your system is supposed to solve. Ask clarifying questions if this is not clear.
- See the interviewer as the customer, requirements might be intentionally vague, and she/he can give you clarifications.
- Write/raise the requirements or assumptions you are making, and base your design on them.
- Feel free to create a diagram if that helps you clarifying your thoughts.

Create a design for a system that fulfills captured requirements (e.g., constraints, scalability, maintenance).

- Things that influence the design typically include: non-functional requirements (amount of load, load distribution, security), ways of interacting with the system (user access, scheduled processes, synchronous/asynchronous communication), and data flow.
- Explicitly mention all the assumptions that are being made.
- Spend majority of time on the Critical Requirements identified and on the core functionality.
- Justify the design choices that are made.
- Design the solution for scale, i.e. would more transactions seamlessly work.
- Research and read through best practices in the tech stack or infrastructure you would like to use. It helps to know how code and hardware can be maintained, scaled, and be made available.

Design for operational performance and plans for failure and can measure the results (e.g., metrics).

- When you have a design that you think solves the customer problem, ensure that it also incorporates operations and think about operations and resilience in your design.
- **For example:**
    - What are the key business and technical metrics for the system? How will someone use that to identify problems?
    - What are the potential bottlenecks or pain points?
    - What failure points exist?
    - What redundancy can we build in to reduce single points of failure?
    - How does someone get logs and debug the system?

Identify potential shortcomings and tradeoffs with different designs (e.g., performance, fault, tolerance, dependencies).

- If there are multiple ways to design a system that satisfy the requirements, describe the trade-offs of both approaches, choose one and explain why. Having a conversation with the interviewer and explaining your thought process, might help you to make a choice.
- We are looking for a high-level understanding of how to design a system and deep knowledge into at least one area. If the interviewer asks you to dive deep into something you are unfamiliar with, tell them and suggest some other area you are familiar with and dive into that.