ChatGPT

Strategies for Senior Backend/SRE FAANG Interview Prep (2025)

Setting a Consistent Study Schedule (2–3 Hours/Day)

Preparing for FAANG-level interviews while working full-time requires consistency and focus. It's crucial to create a realistic weekly plan and stick to it. Aim to dedicate about 2–3 hours each day (around 10–15 hours per week) for focused study [1]. Break your prep into daily chunks – for example, **solve 1–2 coding problems** and spend some time on **system design practice or review** each day [1] [2]. Consistency beats cramming; spreading out practice helps you retain more without burning out [3]. Make sure you **plan ahead what to study** in each session so you don't waste time deciding on the fly [4].

Given your 7 years of experience, *don't over-focus on brute LeetCode grinding at the expense of design and behavioral prep*. One senior engineer reflected that he spent 6 months solving 500 LeetCode problems but then "suffered at behaviorals and system design" – he recommends prioritizing system design and behavioral prep earlier for senior-level roles [5]. In practice, this means allocating some of your daily study time to non-coding topics (design, fundamentals, etc.) rather than only coding problems. It's also wise to **mix topics throughout your prep**; avoid doing only algorithms for weeks then only design later – alternating keeps skills fresh so you don't forget one area while focusing on another [6]. For instance, you might solve a coding problem then immediately spend 15 minutes sketching a design or reviewing a concept. This interleaving strategy helps reinforce learning across domains.

Finally, treat this like a marathon, not a sprint. Maintain healthy habits – take short breaks, get enough sleep, and use weekends for lighter review or an extra mock interview rather than heavy grinding [7]. A steady 2 hours per day over 8+ weeks (≈100+ hours total) is usually sufficient to rebuild interview skills without overwhelming yourself [8] [9]. In fact, one engineer with ~7.5 YOE prepared ~3 months (191 hours total, ~2.1 hours/day) and solved ~100 problems, which was enough to land multiple offers [8]. The key is the **quality** of those hours: stay focused, eliminate distractions, and follow a plan.

Data Structures & Algorithms Practice (Tailored to Python)

For coding interviews, refresh your data structures and algorithms (DSA) knowledge and practice solving problems in Python. As a Python developer, you'll want to be comfortable implementing common algorithms efficiently using Python's features (e.g. know the performance characteristics of lists, dictionaries, heaps, etc. in Python). Start by reviewing core data structures (arrays, linked lists, trees, graphs, stacks/queues, hash tables) and algorithms (sorting, searching, BFS/DFS, recursion, dynamic programming). If it's been a while, spend the first week or two rebuilding fluency in these fundamentals [10] [11] – for example, implement basic structures from scratch and solve a few easy-medium problems on each to ensure you remember how to manipulate them.

Use curated problem sets instead of random questions. A targeted list will cover the most important patterns without wasting time. Highly regarded free "roadmaps" in 2025 include **Neetcode 150**, **Blind 75**, and **Striver's A2Z sheet**, which compile the most common and high-yield LeetCode problems [12] [13]. These sets focus on patterns like two-pointers, sliding windows, binary search, BFS/DFS, dynamic programming, etc., which are the building blocks of many interview questions. It's more effective to thoroughly solve ~100–150 representative problems covering all key patterns than to grind hundreds of random questions [14] [13]. For instance, one plan is to work through ~10–20 important topics and solve ~150–200 medium-level problems in total, rather than trying to brute-force 600 questions [15]. LeetCode is the main platform for practice; if possible, use LeetCode Premium to filter questions by FAANG companies and frequency (so you focus on those that are most relevant) [16]. Since you're using Python, take advantage of Python-specific resources: for example, **Elements of Programming Interviews in Python** (book) or the **NeetCode** website, which offers explanations and solutions in Python. These can help you learn "Pythonic" approaches to problems and common pitfalls. Also consider occasional challenges on HackerRank or CodeSignal, since companies sometimes use those for screening; it will get you comfortable with Python in an online IDE setting.

When practicing coding problems, simulate real interview conditions: use a simple text editor or LeetCode's environment (no auto-complete) to code, and practice explaining your thought process out loud. Focus on writing clean, readable code in Python – interviewer feedback often values code clarity over using obscure one-liners. That said, know your language's strengths: using built-in functions and libraries (like `sorted()` or `heapq`) is usually fine, but be ready to discuss their complexity. Pay attention to Python-specific details (e.g. how default sorting is Timsort O(n log n), or that `list.pop(0)` is O(n) whereas collections.deque is efficient for queue ops).

Routine: Each weekday, you might solve 1 new problem and review 1–2 past problems. Use the weekends for a slightly longer session to perhaps simulate a full coding round or tackle a harder problem. Always **review solutions and patterns** after solving. If you struggle or even if you succeed, read official solutions or top discussions to learn new techniques or optimizations in Python [17] [18]. Take notes on the approaches. Especially note patterns – e.g. recognize that a problem is an application of "two-sum/hash map" or "tree DFS + memoization." Over time you'll see that many problems are variants of classic patterns. Consider using flashcards or an **Anki deck** to remember key ideas and Python tricks (one experienced prepper made ~480 flashcards for algorithms/design concepts) [8]. A spaced-repetition system will ensure you revisit tricky concepts regularly [19] – for example, you might have a card for "BFS vs DFS tradeoffs" or pseudocode for merge sort, etc., and quiz yourself periodically.

System Design Preparation (High-Level Design Skills)

For senior backend roles, **system design interviews** are often the make-or-break round. Since you specifically want to improve here, allocate a significant chunk of your prep to system design. Given your experience, interviewers will expect you to articulate how to design complex, scalable systems and to demonstrate a solid grasp of architectural trade-offs. Current FAANG expectations (for L5 equivalent and above) are that you can discuss **scalability, reliability, availability, consistency, and performance trade-offs** for large systems [20] – essentially, showing that you can **think like an architect**.

Build foundational knowledge first: Spend a couple of weeks revising core system design concepts and modern architectures [21] [22]. Key areas include: load balancing, caching, database scaling (SQL vs NoSQL, sharding, replication), message queuing, microservices vs monoliths, CAP theorem and consistency models, and specific topics like rate limiting, distributed logging, etc. It's helpful to read or watch primers on these topics. For example, the free **System Design Primer** on GitHub is a highly recommended starting point covering the basics (DNS, HTTP, CDN, etc.) [23] . Also consider resources like **Gaurav Sen's YouTube channel** (system design tutorials) and the classic book **Designing Data-Intensive Applications** by Martin Kleppmann (often cited as a must-read "DDIA") [24] [23] . DDIA is heavy, but it deeply explores trade-offs in data systems – if it feels too dense, there are YouTube series (e.g. *"Jordan Has A Solution"* aka *Jordan has no life*) that summarize DDIA's key points in a digestible way [23] .

Study established solutions and patterns: Learn how common systems are designed. Focus on a few **real-world design scenarios** that are frequently asked: e.g. design a URL shortener, a news feed, a messaging/chat service, an e-commerce checkout system, a ride-sharing system, etc. There are many tutorials, blogs, and courses that walk through these examples. ByteByteGo (by Alex Xu) is an up-to-date platform in 2025 that provides visual system design case studies and even covers new topics like designing ML systems or incorporating generative AI [25] . Alex Xu's **"System Design Interview" books (volumes 1 & 2)** remain popular for curated case studies – one roadmap suggests reading **Alex Xu's Book 1 then Book 2** once you have the basics down [23] . Online courses like Educative's **Grokking the System Design Interview** or DesignGuru's system design course are also tailored for interview scenarios, providing a framework to approach any design. The key is to internalize a **framework** for system design: e.g. clarifying requirements, defining API/data models, sketching high-level components, discussing each component's design (database choice, caching, etc.), and addressing bottlenecks or failure points. Practice applying this framework to each scenario.

After learning concepts, **practice designing systems out loud**. System design skill comes from doing, not just reading. Use a whiteboard or pen and paper to simulate the interview: take a prompt (say, "Design Twitter") and time yourself for 30–45 minutes mapping out a solution. Emphasize **communication**: in the interview you should constantly explain your thinking, so practice describing your design rationale in a structured way. Focus on clarity and covering the fundamentals – interviewers don't expect exact figures for every node or an exhaustive low-level design, but they *do* expect you to cover the major components and make reasoned decisions. As one guide notes: *"You're not expected to design Uber in 30 minutes. You are expected to think clearly and communicate tradeoffs."* [26] . This means it's fine if the design isn't extremely detailed, but you should proactively discuss why you choose one approach over alternatives (e.g. SQL vs NoSQL, or why add a cache and how to handle its consistency issues). In fact, for senior-level candidates, being able to **deep-dive on specific components and discuss trade-offs** is what sets you apart [27] . For example, a junior might say "we'll add a cache to improve performance"; a senior should add "...but we must consider cache eviction policies, consistency (cache invalidation), and failure modes like the thundering herd problem" [28] . Practice articulating these kinds of considerations for each design decision.

Leverage the best system design resources (2025): Modern content like **ByteByteGo** (which now covers not only classic system design but also OOP design and even GenAI system design) is highly regarded [25] . Many engineers also use **educative.io's System Design courses**, **Exponent's system design prep material**, or **Design Gurus**. If you prefer books: aside from Alex Xu's books and DDIA, you might explore **"Software Architecture for Developers"** or **"Systems Analysis and Design"**, but those are supplementary. To keep your study efficient, focus on interactive and updated resources. For instance, websites like **hellointerview.io** (mentioned by some candidates) aggregate system design Q&A for quick reference [23] . Community compilations of design Q&A (e.g. a list of 20 common system design questions on a blog [29]) can be useful for quick practice prompts.

Finally, **do mock system design interviews**. This is critical – it's one thing to brainstorm alone, and another to explain to someone and handle feedback. Try to schedule a mock design interview at least every 1–2 weeks during your prep. You can pair up with a colleague or use platforms like **Pramp** (free peer-to-peer mock interviews) [30]. There are also paid services where you can interview with ex-FAANG engineers; for example, **Exponent** or **Interviewing.io** offer mock interviews with feedback [31] [32]. Going through a realistic simulation will highlight areas to improve – maybe you forgot to ask about requirements, or you got flustered on a particular question. Use that feedback to adjust your preparation. With each mock, you'll get more fluent at driving the conversation and covering the key points within the time. The goal is that by the time of the real interview, you've **already practiced "designing X" many times** and can confidently tackle an unfamiliar design by drawing parallels to things you've seen.

SRE-Specific Topics and Scenarios

For Site Reliability Engineer (SRE) roles, the interview scope is broader than algorithms and high-level system design. SRE interviews (at places like Google and other top tech companies) will **still assess coding and architecture skills**, but also drill into your practical understanding of systems, Linux, networking, and troubleshooting under real-world conditions. In Google's SRE interview, for example, candidates are expected to be *"very good at LeetCode-type coding problems, know all about Linux internals, and be excellent at designing systems and working out why they go wrong."* [33]. In other words, preparation for SRE should cover:
- **Data Structures & Algorithms:** proficiency in coding is required (often in a scripting language like Python). Expect at least one coding round where you solve algorithmic problems similar to a SWE interview. Keep practicing LeetCode mediums and some hards to sharpen these skills. SRE coding questions might sometimes skew to practical scripting (parsing logs, automating tasks) in addition to classic algos, so be prepared to write clean, correct code quickly [34].
- **Systems Administration and Linux Fundamentals:** You should review how operating systems work (especially Linux, since SREs deal with production Linux systems). Key topics include processes and threads, concurrency basics, memory management, file systems, I/O, and shell scripting. Be ready for questions like *"What is a zombie process?"*, *"Explain how a TCP handshake works,"* or *"What happens when you type* `ls -l` *in the shell?"* [35] [36]. Use resources like the **Google SRE book (Site Reliability Engineering)** and its companion (The SRE Workbook) to understand concepts like SLIs/SLOs, error budgets, monitoring and alerting, incident response, etc. A popular GitHub repo for SRE interview prep compiles resources on Linux internals, networking, and troubleshooting – covering topics from file descriptors and signals to debugging tools [37] [38]. Make sure you can reason through scenarios such as diagnosing high CPU on a server, a network partition, or a disk filling up.
- **Troubleshooting and Scenario Questions:** SRE interviews often include hypothetical outage scenarios or production problems to solve. Practice a structured approach to troubleshooting: for instance, if a web service is down, talk through checking logs, metrics, recent deploys, dependency health, etc. Interviewers may present a scenario like *"Users can't reach service X – how do you investigate?"* or give an on-call page and ask what you do next [39] [40]. To prepare, it helps to read **post-mortems of real incidents** (Google Cloud and AWS sometimes publish those) or to role-play firefighting an outage. Develop a mindset of forming hypotheses and methodically eliminating causes.
- **SRE System Design (Reliability Focus):** In addition to classic system design, SREs may be asked to do a **Non-Abstract Large System Design (NALSD)** which often means designing a system with reliability in mind. For example, instead of just designing a chat app, you might be asked how to make it highly available across data centers, how to handle failovers, how to deploy it without downtime, etc. When practicing system design, pay special attention to **failure modes and resiliency**: how does the system recover from outages, how to implement monitoring/alerting, how to ensure zero-downtime deployments (blue-green,

canary releases) [41] [42]. Be comfortable discussing concepts like **Service Level Objectives (SLOs)** and **error budgets** (these are core to SRE culture) [43] – e.g., you might be asked how you'd set an SLO for a given system and what you do if it's breached.
- **Automation and Tools:** SREs are expected to automate manual work. You might get a question like *"Write a script to monitor CPU usage and alert if > X%"* or *"How would you automate log rotation/backup?"*. Practice a bit of **Python scripting and Bash** for tasks like parsing text, monitoring system stats, or calling APIs (since your background is Python, you can leverage that for automation). Also familiarize yourself with popular tools (even conceptually) – e.g. containers and Kubernetes, CI/CD pipelines, Infrastructure as Code – as these could come up in conversation.

To cover these efficiently, incorporate a little bit of SRE-specific study into your weekly plan. For example, each week pick one **SRE knowledge topic** (like "Linux networking" or "Monitoring & Alerting") and spend an hour reading up on it or solving a related problem. You could use the **Google SRE workbook exercises**, or try some "war game" style challenges (there are sites like GrepBrick or others that offer Linux troubleshooting challenges). The finalround.ai blog notes that SRE interviews aim to *"assess a candidate's ability to maintain and improve reliability, scalability, and performance of systems"*, covering areas like **system design, incident management, automation, and coding** [44]. So ensure your preparation touches each of those areas. In practice, this might mean: do LeetCode and code problems (coding), read Google's SRE case studies and practice outage scenarios (incident management), automate a small task or at least outline how you would (automation), and do the system design practice as discussed (design).

Measuring Progress and Retaining Knowledge (6–10 Week Timeline)

With ~6–10 weeks to prepare, it's important to track your progress and ensure you're retaining what you learn. Here are some strategies to keep yourself on course:

- **Set Milestones and Use a Tracker:** Break down your timeline into weekly goals. For example: by end of Week 2, complete review of arrays/strings and solve 15 problems; by Week 4, finish at least one pass of system design fundamentals; by Week 6, complete 50 problems and 3 full design practices, etc. Create a simple spreadsheet or use a habit-tracking app to log your daily study. Logging hours and topics keeps you accountable. One engineer shared a **"monthly tracker"** template they used to log hours studied and problems solved, which helped ensure they averaged ~2 hours per day consistently [8]. Seeing the numbers (e.g. total problems solved, total hours spent) will both motivate you and let you adjust if you're falling behind in one area.

- **Spaced Repetition for Retention:** As you accumulate knowledge (be it algorithm tricks or system design concepts), regularly revisit it. Use flashcards or a notes review system. For algorithms, you might make flashcards for patterns or important problems – e.g. one side describes a problem or pattern, the other side has the approach. Quiz yourself on these every few days, increasing the interval if you keep remembering correctly. This technique of **spaced repetition** is proven to lock in memory [19]. For system design, you might maintain a one-page cheat-sheet of key points (CAP theorem, pros/cons of SQL vs NoSQL, etc.) and glance at it often. Another method is to re-design systems you did earlier, from memory – see if a week later you can still outline the design of that URL shortener or whatever you practiced. If you can, that's a good sign you retained it; if not, refresh that topic.

- **Regular Self-Assessment:** Every week or two, do a "mock test" for yourself. This could be a timed coding problem (set 30 minutes and do a medium problem as if it's an interview) or an oral walkthrough of a system design prompt. Afterwards, critically assess your performance. Did you finish in time? Did you get stuck? Checking improvement in these mock scenarios is one of the best ways to measure progress [45] [46]. For coding, track metrics like: problems solved per week, success rate (how often you get it without peeking at solutions), and the difficulty levels you're comfortable with. Ideally, over 6–10 weeks you'll see yourself graduating from struggling on mediums to handling mediums comfortably and maybe even tackling a few hard problems. For system design, you might measure progress by the depth of your discussions: early on you might only manage a simple 10,000-foot overview, but later you find you can naturally dive into specifics and cover more aspects within the time.

- **Mock Interviews and Feedback:** Incorporate at least 2–3 mock interviews into your 6–10 week timeline (perhaps one around week 3 or 4, another around week 6, etc.). This can be with a friend or through a service. The feedback you get is invaluable – it will reveal whether you're improving. For instance, maybe in the first mock you forgot to consider edge cases in a coding problem or you stumbled explaining a past project; in the next mock, you should aim to not repeat those mistakes. Keep a notebook of feedback and **lessons learned from each mock**. If an interviewer or peer says you needed a clearer explanation for something, practice that specifically. Improvement in mock interview feedback (e.g. from "code had a bug and communication was a bit shaky" to "code was correct and you explained well") is a concrete measure that you're progressing.

- **Retention Checks with Anki/Quizzes:** If you use Anki or similar, look at your retention metrics (Anki shows how many cards you recall, etc.). If you notice certain topics consistently slip (say you keep forgetting how exactly consistent hashing works, or the details of a Linux command), that's a sign to give those more attention. You can also do quick quizzes for yourself on fundamentals each week. For example, at the end of the week, try to write down on a blank sheet: all the common time complexities, or the steps of designing a scalable system, or the key points of the STAR method for behavioral answers. Doing these recall exercises ensures the knowledge stays fresh.

- **Reflect and Adjust Weekly:** At the end of each week, take 15 minutes to reflect on what went well and what didn't. Are you weaker in one area than you thought? For instance, maybe you realized graph algorithms are still confusing – then you allocate extra time next week for graphs. The ability to self-correct is important. As one 2025 interview prep guide put it, *"solve, reflect, group by pattern, repeat."* Always loop back and analyze your performance [13]. If you solved 10 dynamic programming problems this week but still feel unsure, maybe next week do 5 more and review a DP tutorial to solidify the pattern. This iterative approach will steadily improve your competence.

By the end of a 6–10 week regimen, you should have a record of dozens of problems solved, a handful of system designs practiced, and multiple feedback points from mocks – all of which give you confidence that you're ready. If possible, **reserve the last few days before interviews for light review and rest** [47]. Use that time to go over your notes, redo a couple of your "favorite" problems and design questions, and mentally rehearse behavioral stories. Avoid cramming anything new at the last minute [48]. If you've followed your plan, you'll be as prepared as you can be – the final step is to enter the interviews well-rested and mentally sharp.

Leveraging Modern Tools and Resources (2025 Trends)

Take advantage of new tools and platforms that can make your prep more effective and efficient. As of 2025, there are several AI-driven and specialized services geared towards interview preparation:

- **AI Coding Assistants and Reviewers:** You can use AI to **accelerate learning and get feedback**. For example, many candidates use ChatGPT (or similar LLMs) to explain tricky concepts, generate examples, or even act as a pseudo-interviewer. If you're stuck on a problem, ChatGPT can hint at a solution or help analyze why your code isn't working. Some platforms specifically target coding interviews – *BugFree.ai* is one such tool that will *"auto-review your code & suggest optimizations"* in practice problems [49] . It's like having an AI code mentor: you write a solution and the tool points out potential bugs or improvements. This can be especially useful for Python, where an AI might suggest more pythonic idioms or flag edge cases you missed.

- **AI Mock Interviews:** Beyond just coding, AI can help simulate interview Q&A. **FinalRound AI** and **Interview Copilot** are examples of services where an AI will ask you interview questions (technical or behavioral) and evaluate your responses. For instance, you could practice answering "Tell me about a time you handled a production incident" to an AI and it will analyze your answer for clarity or missing details. Some AI tools also offer system design scenario walkthroughs. *Codemia* is another emerging platform that provides *"mock system design + DSA with AI feedback"* [50] . It can present you with a design prompt, let you outline your approach, and then give you pointers on what you might improve or what you missed – this is great for honing design communication when you don't have a human partner available. Keep in mind AI feedback isn't perfect, but it's a handy supplementary tool when used wisely (and it's available 24/7).

- **Traditional Mock Interview Platforms:** In addition to AI, **human feedback** is still crucial. Platforms like **Pramp** (free) pair you with another candidate for mutual mock interviews [51] . **Interviewing.io**, **Exponent**, and **Interview Kickstart** are paid options where you can practice with experienced interviewers or coaches. Exponent, for example, has experts for system design and behavioral coaching [52] . If you feel you need extra help with communication or system design depth, a session or two with a professional coach or an ex-FAANG interviewer can be worth it – they can pinpoint subtle weaknesses and also share the latest nuances of what top companies expect. (Many FAANG companies have evolving interview trends, and a coach who interviews candidates regularly can give insight into current question styles).

- **Interactive Learning Platforms:** There are updated courses and interactive platforms tailored for interview prep that can keep you engaged. For coding practice, aside from LeetCode, check out **NeetCode.io** (which offers structured study plans and video explanations), or **AlgoMonster**, which teaches DSA patterns interactively (it was noted for its pattern-based approach in 2025) [53] . Some platforms like **HackerRank** and **LeetCode** have also started integrating AI hints – e.g., LeetCode's Discuss might have GPT-generated explanations for solutions. Use these features if you're truly stuck, but try not to become too reliant on hints (you still need to simulate real interview conditions).

- **System Design Resources (Latest):** 2025 has seen system design content expand into new domains. **ByteByteGo** is frequently recommended – it's an all-in-one site for system design with newsletters, videos, and a structured curriculum, created by an ex-FB engineer [25] . It's updated with examples including modern topics like machine learning system design and even generative AI system components, which could be relevant as some companies now ask about designing AI-driven systems. Also, **DesignGurus** (the creators of Grokking) have a **System Design School** that provides interactive case studies and even one-on-one mentoring options [54] [55]. If you prefer YouTube, channels like Gaurav Sen, System Design Interview by Exponent, and others release up-to-date videos – subscribe to a couple so you get fresh perspectives.

- **Community and Forums:** Don't overlook communities like **r/leetcode** and **Blind** – in 2025, candidates often share their interview experiences, and you can glean insights about the latest interview trends at specific companies. For example, some Reddit threads discuss how certain companies might emphasize one area over another. Just be careful to avoid getting overwhelmed by others' panic; use community info constructively (e.g., a popular post might have a cheat-sheet of "Top 10 system design concepts for 2025 interviews" – that's useful). Also, consider joining or forming a small study group with peers. Even one or two study sessions a week where you and a friend ask each other a coding question or design prompt can boost your preparation and keep you accountable.

In summary, **make 2025's tools work for you**: use AI to speed up feedback loops (but still do the thinking yourself), use mock services to simulate pressure, and use curated content to stay aligned with what FAANG companies expect. The bar for senior backend and SRE roles is not just solving problems, but demonstrating leadership in problem-solving – clear communication, thoughtful design decisions, and the ability to handle complex scenarios calmly. By following a structured study regimen, practicing across coding, design, and SRE topics, and leveraging modern prep tools, you'll be aligning your preparation with *current* FAANG standards. Remember that interview prep itself is a process of continuous improvement. Stay consistent, measure your progress, and adjust as needed. With 7 years of experience, you likely have a wealth of knowledge to draw on – this preparation will be about packaging that knowledge to meet FAANG's interview format. Good luck, and happy studying!

Sources: Recent interview prep guides and expert advice have informed these strategies. For example, a 2025 FAANG prep guide emphasizes mastering DSA and system design with top-notch resources (NeetCode, Grokking, ByteByteGo, etc.) [12] [56], and suggests daily practice goals like *"2 coding questions + 1 design problem"* for several months [2]. Senior candidates on forums stress the importance of communication and system design depth – *"People want conciseness… ability to communicate as a senior engineer"* and solid prep via system design primers and Alex Xu's books [57]. An experienced engineer's post underscored mixing LeetCode practice with system design so neither is neglected [6], using Anki/spaced repetition for retention [58], and doing mocks to gauge readiness. Furthermore, Google's SRE interview guidelines highlight needing strong coding skills, Linux internals knowledge, and an aptitude for troubleshooting complex systems [33] – reflecting why our plan incorporates OS and networking review for SRE. Finally, new AI-driven tools (e.g. BugFree.ai, Codemia) and mock platforms (Exponent, Pramp) have been recommended by interview coaches as of 2025 to give candidates an edge with instant feedback and realistic practice [59] [49]. All these informed the comprehensive approach above, tailored to the expectations of today's top tech firms.

---

[1] [14] [31] [32] [45] [46] Complete Coding Interview Preparation Roadmap for 2025
https://www.designgurus.io/blog/coding-interview-prep-roadmap-2025

2 12 16 19 20 24 25 29 30 49 50 51 52 53 54 55 56 59 How to Prepare for FAANG Interviews in 2025(With Top Free and Paid Resources) | by javinpaul | Javarevisited | Jul, 2025 | Medium
https://medium.com/javarevisited/how-to-prepare-for-faang-interviews-in-2025-with-top-free-and-paid-resources-3275c546724d

3 9 10 11 13 26 47 48 How Long to Prepare for Coding Interviews? | by Kei Zee | Jul, 2025 | Medium
https://medium.com/@keizee01119/how-long-to-prepare-for-coding-interviews-5960c9e4ecb4

4 6 7 8 17 18 58 A detailed interview prep guide for experienced devs : r/leetcode
https://www.reddit.com/r/leetcode/comments/1jffkq0/a_detailed_interview_prep_guide_for_experienced/

5 15 23 27 28 57 How are you planning on studying for interviews in 2025? : r/leetcode
https://www.reddit.com/r/leetcode/comments/1htl6d6/how_are_you_planning_on_studying_for_interviews/

21 22 How long to prepare for a system design interview?
https://www.designgurus.io/answers/detail/how-long-to-prepare-for-a-system-design-interview

33 34 35 36 39 40 Google Site Reliability Engineer (SRE) Interview (questions, process, prep) - IGotAnOffer
https://igotanoffer.com/blogs/tech/google-site-reliability-engineer-interview

37 38 GitHub - mxssl/sre-interview-prep-guide: Site Reliability Engineer Interview Preparation Guide
https://github.com/mxssl/sre-interview-prep-guide

41 42 43 44 25 Essential SRE Interview Questions You Need to Know
https://www.finalroundai.com/blog/sre-interview-questions