Engineering Career Skills (2024 Prototype)

This is a work-in-progress example of a career development spreadsheet being explored at OpenSesame. It's being shared for the purpose of soliciting feedback only.

This document is meant to accompany the Engineering Career Development spreadsheet. It explains the skills listed in that sheet.

The descriptions in this document *are not comprehensive.* Your manager has final say over what it means to be fluent in any given skill. If you feel their definition is incorrect or unfair, speak to their manager.

All Engineers

These skill sets represent core skills. All engineers are expected to have these skills, or to become fluent shortly after joining OpenSesame.

Professionalism (All)

Spoken and written English

You understand your coworkers and can usually express yourself without searching for words or phrases.

Excluded: Specialized types of communication, such as technical writing, are separate skills.

Work ethic

You work the full hours expected of you (typically 40 hours per week, plus time for lunch). While working, you give your work your full attention, without looking for excuses to procrastinate or delay. You work within the standard hours of your team, which are defined by your manager.

Intrinsic motivation

You are motivated by the enjoyment of engineering. This includes things like solving challenging problems, creating valuable products, making users' lives better, working with smart people, and learning new technologies. Although you expect to be compensated fairly, you don't need the promise of rewards or the threat of punishment to do your best work.

Excluded: You aren't expected to enjoy every aspect of your job.

Remote attendance

You have the equipment, Internet bandwidth, and knowledge to use your team's communication tools, such as Zoom, Google Docs, Miro, and Confluence. You are fully present for synchronous sessions, such as Zoom meetings: joining on time, paying attention, and avoiding distractions. Your camera is typically on during video calls. You are responsive to asynchronous communication, such as email and Slack, according to the norms set by your team and your manager.

In-person attendance

You attend mandatory in-person events. You make arrangements for child care, pet care, and other necessities that enable you to travel away from home. You work with your manager to book and reimburse flights and other aspects of required travel. At events, you are aware of schedules, arrive on time, and participate fully.

Active participation

During group discussions, you're fully present and contributing. You share your thoughts while leaving space for others to speak.

Respectful communication

You treat people with respect. When you disagree with someone, you express yourself calmly and rationally, avoiding personal attacks. When you find yourself in a personality conflict, you ask your manager or a trusted colleague to help mediate. If a conflict doesn't go your way, you give the ultimate decision your full support, and you don't go behind the other person's back to sabotage or complain.

See also: Defend a contrary stance

Transparency

You make your work, plans, and progress visible to your team. When something is taking longer than expected, or otherwise isn't going as you hoped, you ensure that the people depending on that work know what's happening so they can adjust. You resist the temptation to hide, delay, or "spin" disappointing news.

Team orientation

You prioritize the productivity and success of your team over your individual tasks. Sometimes that involves helping others rather than finishing your own work; at other times, it involves asking others to help you. You work collaboratively with your team to understand what you all need to do and make plans for completing it together.

Follow the process

You follow the process and norms that your team has agreed upon, within the constraints management has set. However, you don't follow the process blindly: you exercise judgment and work with your team to change the process, or agree to exceptions, when the process doesn't make sense for your situation. You make those changes collaboratively and constructively, not unilaterally, and you check with management when those changes could exceed your team's authority.

See also: Critique the process; Circles and soup

Grit

When faced with a setback or challenge, you don't give up or allow yourself to be distracted. Instead, you rise to the challenge, exploring options and trying ideas. You admit when you're having trouble, and ask for help, rather than "going dark" and pretending everything's fine.

Excluded: Sometimes you need to take a break or find someone else who's better suited to your task, and that's okay. You do so collaboratively and positively, working with your team to find the best way to achieve success.

Absorb feedback

When you’re given feedback about something you’ve done, especially by a manager or mentor, you consider how to change your behavior. If you have doubts, you ask for second opinions rather than dismissing the feedback entirely.

Excluded: It can be hard to hear feedback, especially negative feedback. You aren’t expected to absorb it immediately. It’s okay to let it percolate for a day or so, and the changes you make may not be visible for longer than that.

Growth mindset

You understand that your abilities can be developed through dedication and hard work, and you constantly seek out opportunities to learn and grow. When you discover something that you don’t know, you see it as an opportunity rather than a threat. When you encounter someone who has skills or knowledge you don’t, you look for ways to learn from them rather than feeling inferior or jealous. You’re open about your mistakes and skill gaps, and use them as opportunities to improve and ask for help.

OpenSesame Qualified

You’re either OpenSesame Qualified, or working towards achieving OpenSesame Qualified. You’re expected to be qualified by the end of your first year.

Classroom Engineering (Associate SE)

Object-oriented programming language

You can program in an object-oriented programming language of your choice (or a multi-paradigm language with object-oriented features). You use variables, conditionals, classes, methods, and so forth. You are able to build small programs to solve classroom-style exercises.

Excluded: You aren’t expected to know every detail of your chosen language, just the basics.

Pairing/teaming driver

You participate in pairing and teaming sessions as the person at the keyboard. You work with your partner(s) to take direction on what to program next, suggest ideas, and ask for clarification when you don’t understand. You’re careful to keep them involved in the process and not just treat them as a passive observer. You switch roles frequently, giving them the opportunity to drive as well, according to the norms your team has established.

Excluded: This skill is about being a participant, not a leader. Programming skills are separate.

See also: Pairing/teaming navigator; Pairing/teaming instructor

Classroom-level debugging

Given a small, self-contained, classroom-style program with an error in it, you find the source of the flaw and fix it.

Function and variable abstraction

You break long functions and methods into small pieces with clearly-defined responsibilities, parameters, and return values. You give them names that allow their behavior to be inferred without reading the body of the code, and you avoid unnecessary side effects. You use variable names and constants to make the implementation of your code more clear.

See also: Class abstraction; Codebase design

Software Engineers

These skill sets represent the basics needed to contribute to OpenSesame code without guidance. They’re required at the Software Engineer level and above.

Basic Communication (SE)

Collective ownership

You see your code (and other artifacts) as owned by your *team*, not any one individual. When you have the opportunity to improve it, you do so, coordinating with the rest of your team as necessary. You work without ego, taking pride in your team's ability to improve code you originally wrote, and not trying to take individual credit for any part of the system.

Defend a contrary stance

When you disagree with a suggestion someone on your team has made, you express your disagreement, both in individual and group settings. You are polite but firm in making your objections clear. You give the other parties the opportunity to share their views, you listen and absorb their perspective, and you work together to come to a reasonable conclusion.

Excluded: Defending contrary opinions within groups of people you don't know well is a more advanced skill.

See also: Psychological safety

"Yes, and…"

When you see a flaw in a proposal, you build on it rather than tearing it down. In improv, this is known as "yes, and." For example, if somebody makes a proposal that doesn't consider error conditions, you might say, "yes, and we can add error handling for conditions X, Y, and Z," rather than saying, "no, that doesn't consider error conditions." You work with each other's ideas to create a gestalt plan that is greater than the initial proposal.

Try it their way

When you have a difference of opinion about how to approach a problem, and you and the other party are unable to come to a mutually agreeable decision, you try it their way as an experiment. You work with them to make the decision reversible, come up with a mutually agreeable test, and work together with the sincere intention of making the test succeed.

Technical feedback

You evaluate the merits of a technical direction and provide feedback and suggestions for improvements. When working in your areas of specialty, you take a leadership role in providing feedback and guidance without being asked.

Excluded: This only applies to technical situations within your skillset.

Active listening

When others speak, you aren't just waiting for your turn to speak. Instead, you're listening to truly understand what they're saying, why they're saying it, and how it makes them feel. You engage in a two-way dialog to reach this understanding, and you spend as much effort on understanding their point of view as you do on having them understand yours.

As-built documentation

Your system has a set of technical documentation that describes how it works from a high level, known as "as-built documentation." After making a change, you update the documentation accordingly, when needed. Your changes are clear and readable, strike the correct balance between abstraction and detail, and fit seamlessly into the structure and style of the existing documentation.

See also: Runbook documentation

Basic Leadership (SE)

Basic facilitation

You facilitate well-defined, low-stakes sessions involving people you're comfortable with. Examples include the FaST meeting, standup meetings, and the planning game. When you facilitate, you make sure everyone has a chance to speak, the agenda is followed (or deliberately modified), and activities are parallelized when appropriate.

Excluded: This skill doesn't require you to facilitate sessions involving strong disagreement, unfamiliar people, or unfamiliar agendas.

See also: Intermediate facilitation

Team steward

You volunteer to lead a FaST team. During the FaST meeting, you describe the work to be done and ensure the team has the skills it needs. During the FaST cycle, you help the team follow its process, including facilitating stand-up meetings and planning game sessions, and you ensure everyone has the opportunity to contribute equally.

Further reading: Team Steward Role

Valuable increment steward

You volunteer to be the engineering contact for a valuable increment. You help the team understand the valuable increment, provide a through-line for the team, and help keep stakeholders informed.

Further reading: Valuable Increment Steward Role

Scut work

Every team has work that no one enjoys doing. You volunteer to take the lead on your fair share of that work, without dragging your feet or needing to be reminded.

Excluded: This skill is about stepping up and volunteering. Assigned tasks such as the on-call rotation don't count.

Basic Product (SE)

Your team's product

You understand every aspect of how to use your team's product.

Your team's customers and users

You understand how your team's product is sold, who buys it, who uses it, and why.

User story definition

You work with product managers (PMs) and other members of your team to break valuable increments into stories. You ensure that engineering needs are considered while working with your PMs to ensure they are expressed in terms of stories your PMs understand and value. You work with senior engineers to create stories that are "vertical slices" of the product that the whole team can tackle together, rather than simply rephrasing technology-centric "horizontal slices" that focus on a particular technical area.

See also: Vertical slices

Further reading: The Planning Game

Basic Implementation (SE)

Your team's programming language

You are able to program in the main programming language(s) your team uses.

Excluded: You aren't expected to understand every aspect of the programming language, just the parts your team commonly uses.

See also: All of your team's programming languages

Your team's codebase

You are able to modify and maintain your team's main codebase(s). You aren't an expert in every aspect of the codebase; instead, you specialize in a subset of the areas your team commonly works within. You have enough generalist understanding that you are able to be effective, with help, when working outside your specialty.

When working in your areas of specialty, you take a leadership role and guide people who are unfamiliar with that code. When working outside your areas of specialty, you follow the guidance of people who understand that part of the code better.

See also: All of your team's codebases; Codebase specialty

Basic test-driven development

You apply the "red-green-refactor" cycle of test-driven development (TDD). You work in small steps to form hypotheses, test those hypotheses, and improve the code. You are able to break a problem down into bite-size pieces (typically less than five lines of code each), start small, and incrementally layer on more sophisticated code. You organize your tests so they act as documentation, telling the story of the behavior under test.

For code with an established approach to TDD, you use those patterns to write all production code using TDD. For code without an established TDD pattern, you ask more experienced engineers for help. When uncertain about what code you need to write, you create a throwaway spike solution to gain understanding, then rewrite it with TDD.

Excluded: Refactoring and spike solutions are separate skills.

See also: Sociable unit tests; Narrow integration tests; Retrofitting tests; Spike solutions; Method and variable refactoring; Cross-class refactoring

Sociable unit tests

You use TDD to write sociable unit tests for code that doesn't have direct dependencies on external systems. Each one validates a narrowly-focused part of the system's overall behavior. (That's the "unit" part.") The code under test executes its real dependencies, not test code such as a mock. (That's the "sociable" part.)

Your tests focus on the behavior of the code under test, not its implementation. They *don't* break when the implementation changes, and they *do* break when the behavior under test changes. They also don't break when other parts of the codebase change, either in behavior or implementation, so long as the behavior under test remains unchanged.

Excluded: You're only expected to write sociable unit tests when there's an established pattern for you to follow.

See also: Basic test-driven development; Retrofitting tests

Narrow integration tests

You use TDD to write narrow integration tests for code that has direct dependencies on external systems. (These will typically be infrastructure wrappers.) Each test is narrowly focused on part of the code's dependency on an external system. You test your code's use of the dependency for real, when possible, within the constraint that your tests run entirely locally, with no network calls. When you're unable to run the dependency locally, you simulate it as realistically as possible, subject to normal engineering tradeoffs. You augment your tests with "paranoiac telemetry" that alerts the team when the external systems' behavior changes at runtime.

Excluded: You're only expected to write narrow integration tests when there's an established pattern for you to follow. Paranoiac telemetry is also excluded; it's a separate skill.

See also: Basic test-driven development; Simple dependency integration; Complex dependency integration; Paranoiac telemetry

End-to-end tests

You write tests that confirm a large part of your system works from end to end, starting at the UI (or just below the UI), down to the database and external systems, and back. You use end-to-end tests as a safety net, not as your primary way of catching mistakes. You keep the number of end-to-end tests to a minimum, relying on sociable unit tests and narrow integration tests for the bulk of your testing instead. When those tests *don't* fail and the end-to-end tests *do*, you see it as a failure of the other tests. You analyze the gap in test coverage and work with your team to fill those gaps. Over time, you reduce the number of end-to-end tests required in your team's code to the minimum possible.

Excluded: You're only expected to write end-to-end tests when there's an established pattern for you to follow.

See also: Sociable unit tests; Narrow integration tests

Manual validation

You manually check that your code works the way you think it should, and you confirm with stakeholders as appropriate to confirm your assumptions about how the code should work. This validation doesn't take the place of automated tests, but instead is used to double check that your automated tests are working the way you think they are.

See also: Exploratory testing

Spike solutions

When faced with a problem you don't understand, you create a "spike solution" to explore the problem. The spike solution is throwaway code, typically a standalone program, that ignores engineering practices and does the bare minimum needed to help you understand how to solve the problem. Work on the spike is typically timeboxed to less than a day or two. Once you have the understanding you need, you start over with production-grade code using proper software engineering practices.

Basic SQL

You use SQL to perform CRUD operations (create, read, update, and delete). You use inner and outer joins to query data from multiple tables. You are able to execute statements that perform basic database manipulation, such as creating tables, adding columns, changing constraints, and so forth.

Excluded: Database performance optimization is a separate skill.

Pairing/teaming navigator

You participate in pairing and teaming sessions as the person suggesting what to do next. As your partner drives, you consider tactical questions, such as the gaps that exist in the current implementation and what test to write next, as well as strategic questions, such as how the code you're changing fits into the overall design of the system and how to improve it. You keep notes on the changes you'd like to see, and you're ready with the next step when the driver pauses. You find the appropriate balance between letting your driver figure out implementation details and keeping things moving forward.

Excluded: This skill is about being a peer in a pairing or teaming session. Teaching a less experienced partner is a separate skill.

See also: Pairing/teaming driver; Pairing/teaming instructor

Basic algorithms

You're able to solve the sorts of simple algorithmic problems that come up in day-to-day software development, without resorting to web searches or AI. Examples include parsing data out of a string, finding the common elements between two lists, writing a recursive function, and so forth.

Basic performance optimization

You understand the first rule of performance optimization ("don't") and the second rule ("in profiling we trust"). You prioritize clear, simple code over performance tricks, and only apply performance optimizations to code that has been proven to be a bottleneck. You ignore micro-benchmarks in favor of whole-system profiling. You understand the high-level factors gating performance in your codebase, based on information provided by more experienced engineers (specific database calls, service dependencies, algorithms, etc.). You use big-O notation to describe time complexity.

You also work with product managers, when needed, to help them define valuable increments for improving performance. You help them identify the business goal they're trying to solve and define the minimum criteria (such as latency and throughput) needed for success, as well as the maximum criteria at which further optimization has no business value.

Excluded: This skill is about understanding the big picture of optimization and avoiding common optimization mistakes. Profiling systems and improving performance are more advanced skills.

See also: Code performance optimization

Debugging your team's components

You're able to find and fix errors in your team's code, once the problem has been narrowed down to a single process.

Simple dependency integration

Given a third-party dependency with a straightforward API, you implement the dependency into your team's code, using techniques such as infrastructure wrappers (aka Gateways) to minimize the team's ongoing maintenance costs.

See also: Complex dependency integration

Unhappy path thinking

When implementing code, you think about the ways it could fail as much as how it should succeed. You program defensively, considering user error, failures of external systems, and ways your code could be called incorrectly. You find the right balance between handling errors and keeping your code straightforward and easy to understand.

See also: Fail fast

Basic Design (SE)

Decompose problem into tasks

You analyze a problem, such as a user story, and break it down into the specific tasks that you and the other members of your team will need to perform to solve it. You organize the tasks into a plan, such as a task map, that shows opportunities for parallel work and the order in which the tasks should be performed. Each task is less than two hours of work. You update the plan regularly as your understanding of the situation changes.

When the problem involves unknowns, your plan shows the tasks that will be performed to address the unknowns. You include timeboxes, where appropriate, to prevent tasks from spiraling out of control.

Excluded: This only applies to problems within your skill set.

See also: User story definition; Spike solutions

Class abstraction

You organize code and data into classes (or modules, when appropriate), each with a single, clear responsibility. You use encapsulation to hide the internal implementation of your classes from callers, and your public methods represent callers' needs rather than the implementation's needs. You group data together with the code that operates on it. You avoid mutable state and side effects when possible.

See also: Function and variable abstraction, Codebase design

Mental model of your team's codebase

You understand how your team's code fits together at a high level. You're able to draw a system diagram showing how the various components (processes and services) fit together, and you're able to diagram the high-level relationships between important classes and modules in the components your team owns. When a change is needed, you know which parts of the code are likely to need work, and when an issue occurs, you know which parts of the code to investigate. You use that understanding to intuit the behavior of classes and methods from their names alone, when they're well-named, without reading their code.

Excluded: You aren't expected to know the details of every part of every codebase your team is responsible for, but you are expected to understand the high level relationships in all the code your team works with regularly.

Mental model of a complex dependency

You have a deep understanding of the key concepts, abstractions, and metaphors required to work with one of your team's complex third-party dependencies. This goes beyond understanding how to use the dependency; it involves understanding the *why* of how the dependency is used. For example, with React, you would understand what components are, what hooks are, how state is stored, what the virtual DOM is and why it's used, and how these things can lead to problems with retrieving state.

Excluded: You only need to have this level of understanding for one dependency. You also aren't expected to understand the inner workings of the dependency or every detail; just the user-level abstractions and concepts that relate to the way your team uses the dependency.

Method and variable refactoring

As you work with methods and variables as part of your everyday work, you identify ways to make them simpler and easier to understand. You use refactorings such as inlining, extraction, and renaming to implement those improvements whenever it makes sense. Typically, that's multiple times per hour.

See also: Cross-class refactoring; Architectural refactoring

Campsite rule

You never leave the code worse than you found it. Additionally, as you work, you're constantly looking for opportunities to make the code simpler and easier to understand. Some might be as simple as renaming a variable or factoring out a method. When you see a refactoring that's easy to make, you make it.

When you see opportunities for bigger improvements, you consider your weekly improvement budget, which is defined by your manager. (There's no need to track your time formally; gut feel is enough.) If there's enough time remaining, you identify an incremental change that won't take more than an hour or two. If there isn't enough time remaining, or if you don't know how to make a useful incremental change, you let it go.

Your campsite improvements always relate to the code you're currently working on. If an improvement isn't directly related, or now isn't a good time to make the change, you let it go. (You might make a note to work on a change at the end of your current task, but no longer than that.) There's infinite opportunity for improvement, and you focus on the opportunities in front of you, not ones you saw in the past.

You don't batch up improvements for another day or put them on a team "to-do" list. As Yoda said: "Do, or do not. There is no //TODO."

See also: Risk-driven architecture

Fail fast

As you implement functions and methods, you think about how they could be called incorrectly, and how your assumptions about the internal state of your system could be incorrect. You use run-time assertions to cause your code to fail in a safe and unambiguous way when those expectations aren't met. You strike the appropriate balance between catching errors and keeping the code clear and readable.

Excluded: This skill is about catching internal programming errors, not external systems.

See also: Unhappy path thinking; Paranoiac telemetry

Paranoiac telemetry

As you write code that interfaces with external systems (such as services), you think about how those systems could behave differently than you expect. You assume that the external system will fail some day, for reasons such as intentional changes, outages, or poor throughput. You work with your customer representatives (such as UX designer and product manager) and other engineers to decide how to handle those errors and program your code to fail safe.

You work with your team to ensure error handling code is cohesive, not spread throughout the system. Your team strikes the right balance between handling errors at the point where they happen and delegating them up the call chain to a part of the code that has more context about how to handle the error. If logging and alerting is required (and it usually is), your team avoids spamming log messages throughout the system; instead, your logging and alerting is designed to provide clear, concise information that avoids false positives.

Excluded: This skill is about dealing with external systems, not internal programming errors.

See also: Unhappy path thinking; Fail fast; Narrow integration tests

Evaluate simple dependencies

Given a need for a third-party dependency with a straightforward API, you evaluate options and choose one that best suits the team's needs. You consider factors such as fitness for purpose, future needs, simplicity, cost, maintainability, support, and longevity. You trade off the costs of using the dependency versus the team building and maintaining something itself. As you consider buy-vs-build tradeoffs, you focus on the cost to build just the parts the team needs, not the cost of replacing the entire dependency.

See also: Evaluate complex dependencies

Basic Operations (SE)

Source control

You use git for everyday operations, such as checking out, staging, committing, amending, and reverting; tagging, branching, merging, and rebasing; and pushing and pulling. You have a mental model of how these commands manipulate git's underlying commit graph. You are able to view commit history, view the commit graph, and use bisecting to locate bugs. You follow your team's source control strategy, particularly around branching and integration.

Excluded: You aren't expected to understand git's internals or be able to fix the repository when something goes awry.

Your team's release process

You captain a release from start to finish, according to the process defined for your team. You ask your team for help when you run into obstacles that you cannot solve on your own. You are proactive in raising any concerns regarding the release to your team and manager.

Further reading: Legacy release documents are located in Testing Environment Process and Release Responsibilities.

On-call responsibility

When on call, you monitor on-call communication channels (such as alerts and Slack) and respond within the timeframes established by your team and manager. You understand your team's on-call procedures, including incident response, and are able to execute them promptly and correctly.

On-call triaging

When on call, and you learn about a possible issue, you work with the person who reported it to determine the severity of the issue. You bring in other engineers, as needed, to help you understand the issue. You decide on how the issue should be handled, according to your team's standards and the issue's severity (ranging from "non-issue" to "declare an incident") and make sure appropriate followup actions are taken.

See also: Incident leader

Issue investigation

You are familiar with your team's observability tools, such as New Relic and logs, and use them to investigate production issues. You demonstrate grit and persistence in investigating issues, some of which may not be easily reproduced.

See also: Observability; Incident fixer

Your team's cloud infrastructure

You modify and maintain your team's cloud infrastructure.

Code vulnerability awareness

When writing code, you consider security vulnerabilities and write your code to prevent them. Examples include trust boundary violations, injection attacks, and URL manipulation. You understand the common ways code can result in vulnerabilities and take care to avoid them.

Cloud vulnerability awareness

When configuring cloud infrastructure, you follow the team's documentation on ensuring a holistic security stance, leaving no ambiguity as to what layer is responsible for what. If the documentation is incomplete or unclear, you work with senior engineers to improve it.

Excluded: This skill is about following documented processes, not inventing or improving them.

Senior Software Engineers

These skill sets represent the foundations required for being a peer leader at OpenSesame. They're required at the Senior Software Engineer level and above.

Advanced Communication (Senior SE)

Clear and concise speaking

In off-the-cuff conversation, you organize your thoughts on the fly and present them in a way that’s clear and easy to understand.

Excluded: This skill is about participating in conversations. Public speaking and preparing presentations are separate skills.

Clear and concise writing

Your writing is organized and makes its points clearly. You avoid complicated language and unnecessary words.

Technical diagramming

You use technical diagrams to make complicated ideas clear, both on the fly and prepared in advance. You keep your diagrams simple and clear, avoiding extraneous details. You’re comfortable with remote tools, such as Miro, as well as in-person tools, such as a whiteboard.

Explain mental model

You provide clear explanations of systems you understand well. (For example, you might explain part of your team’s codebase or one of your team’s dependencies.) Your explanations use a mix of demonstrations and diagrams to help people understand how those systems fit together, so they can work on them without your help.

Excluded: This skill only applies to technologies that you understand well.

Ensure everyone’s voice is heard

In group conversations, you’re aware of who’s speaking and who’s not. If someone is being ignored or overridden by louder voices, you help amplify their thoughts, without taking credit for their ideas. You draw people into the conversation without pressuring them or putting them on the spot.

Coalition building

When someone has an idea that's worth promoting, but might be unpopular or difficult to understand, you help them create a coalition around the idea. You socialize the idea in individual and small group discussions, respond to feedback, and help them refine the idea. When they're ready, you help them present the idea, with the support of the coalition, to its ultimate audience. You're careful not to steal the spotlight; your contributions amplify their voice rather than your own.

Interpersonal feedback

You give people feedback on their interpersonal behavior. Your feedback emphasizes how their behavior affects you, without passing judgment on them as a person. You share positive feedback publicly or privately, and critical feedback only privately. Your feedback is respectful, clear, and actionable. You don't shy away from providing feedback, and you do so without being rude or damaging relationships.

Further reading: Interpersonal Feedback

Runbook documentation

Your system has a set of technical documentation that describes how it works in production, known as a "runbook." As your team makes changes, you ensure that documentation is kept up to date. Your changes are clear, readable, and fit seamlessly into the structure and style of the existing documentation.

See also: As-built documentation

Advanced Leadership (Senior SE)

Peer leadership

In any given situation, you follow the lead of the person with the most knowledge of the situation at hand. (This is Mary Parker Follett's "Law of the Situation.") You're adept at taking the lead *and* following the lead of others.

You work with your manager to identify the peer leadership roles that fit your skills, personality, and needs of the team best. You exercise those roles frequently while leaving space for others to exercise leadership roles themselves.

Further reading: Leadership Roles

See also: Leaderful teams

Comfort with ambiguity

You accept that the world is inherently uncertain and no answer is correct for all situations. You make decisions based on the context of the specific situation you're facing, rather than absolutist "right way" and "wrong way" judgments. You make decisions at the "latest responsible moment," even if important details are still unclear, rather than locking down decisions early or waiting too long for every detail to be certain. You expect your plans to change and actively seek out opportunities to improve them.

Risk management

When making plans, you consider how things might go wrong or need to change. You design your plans to fail fast (expose errors early) and you avoid irreversible decisions. Your plans favor providing value incrementally, so you can achieve partial success even if something goes wrong, and you prepare contingency plans for particularly risky scenarios. You avoid "big bang" plans, instead taking small steps that can be proven as you go, preferably "for real" in production.

Intermediate facilitation

You design and facilitate medium and high-stakes sessions involving people you're comfortable with, such as retrospectives and incident analyses. When the session format isn't already defined, you create the agenda using your toolkit of activities and facilitation techniques. You ensure your sessions stay on track (or are deliberately modified), everyone has the opportunity to speak, and people remain respectful towards each other, even when they have strong opinions or disagreements. Your sessions are interesting and effective.

Excluded: This skill doesn't require you to facilitate sessions involving people you don't know well.

See also: Basic facilitation

Mentoring and coaching

You provide mentoring and coaching to people with less experience than you. This mentoring takes multiple forms, including one-on-one discussion, but predominantly focuses on hands-on skill development and leading by example. You use your team's real-world work to help your mentees develop their skills. As you do so, you mix theoretical foundations with hands-on work, ensuring that your mentees truly absorb the material and develop their skills, rather than just watching you work.

See also: Leadership specialty

Critique the process

You view your team's process with a critical eye, looking for opportunities to simplify, eliminate waste, and improve team members' satisfaction and productivity. You're an active participant in retrospectives. Outside of retrospectives, you're constantly proposing small tweaks and experimenting with new ideas, within the bounds of your team's authority. When an experiment works out, you share it with other members of your team, and you encourage others to experiment and make changes as well.

See also: Follow the process; Circles and soup; Impediment removal

Circles and soup

You effect change both inside and outside the limits of your team's authority. You think in terms of circles of influence: the things that you control directly, the things your team controls, the things your team can influence, and the "soup" to which the team can only control its response. You use your understanding of those circles to take action that's as effective as possible, even when responding to situations that are outside of your team's authority.

See also: Critique the process; Impediment removal

Advanced Product (Senior SE)

Ownership

You have a sense of pride and ownership in your team's product(s). Rather than just "taking orders," you're proactive about working with your team and stakeholders to understand what you're delivering, why, and how it can be improved. You understand that development capacity is always limited, and OpenSesame has to balance internal needs, business needs, buyer needs, and user needs. You take a pragmatic and collaborative approach to balancing those needs.

Vertical slices

When breaking valuable increments into stories, you help your team create “vertical slices” of the product that the whole team can tackle together. A *vertical slice* is a story that cuts across all layers of the application (front end, back end, database, services, etc.) to make progress towards the valuable increment. In the best case, the increment could be released after each vertical slice, and be usable, if not necessarily marketable.

Excluded: You aren’t expected to be able to create vertical slices in every case, or to always achieve the best case of vertical slices that are releasable.

See also: Simple design; Incremental design; Incremental codebase architecture

Cost/value optimization

You work with product managers and other members of your team to optimize valuable increments’ cost/value tradeoffs. You work with your product manager to understand the underlying value of each valuable increment, then think about how different technical approaches could achieve the same value for less cost. Similarly, you consider the ultimate business purpose of each valuable increment and think about how different technical approaches could achieve more value. You participate in constructive conversations about these opportunities.

Advanced Implementation (Senior SE)

All of your team’s programming languages

You’re able to program in all of the programming languages your team uses, at least at a basic level. You have deep knowledge of the main language(s) your team uses, including some esoteric details. However, you avoid using those details except when it has a major benefit; your code is typically straightforward and readable, even for junior members of the team.

Excluded: You aren’t expected to know every last detail.

See also: Your team’s programming language

All of your team’s codebases

You are able to modify and maintain all of your team’s codebases. You aren’t an expert in every part of every codebase, but you’re effective when working with someone who is. You have a mental model of how everything fits together that allows you to find your way around when an expert isn't available.

See also: Your team's codebase; Codebase specialty

Codebase specialty

You have deep expertise in a subset of your team's codebases and languages. (You decide on the specific subset with your manager.) Your team recognizes your expertise in that subset and turns to you for help and advice when working in that area.

See also: Leadership specialty

Code performance optimization

You use profiling tools to understand the performance bottlenecks in your team's code. When a bottleneck is found, you perform targeted optimizations to improve performance of the overall system. You prefer macro-optimizations over micro-optimizations. For example, you might reduce usage of expensive resources or improve the big-O time complexity of an algorithm.

Optimizations take time and often make code harder to understand, so you only profile and optimize based on proven needs. You revert optimizations that don't significantly improve overall system performance unless they make the code easier to understand. You only optimize to the degree needed to satisfy your performance goals.

Excluded: This skill is specific to application code. System performance analysis, distributed system optimization, and database optimization are separate skills.

Complex dependency integration

You help your team adopt third-party dependencies with complex APIs. For each new dependency, you start by developing a mental model of the dependency. You do so with activities such as reading documentation, walking through tutorials, and creating spike solutions.

Once you have a mental model of the dependency, you implement it into your team's code. You use techniques such as infrastructure wrappers to minimize the team's ongoing maintenance costs. You teach your mental model to other members of the team so they can use and maintain the dependency as well.

See also: Simple dependency integration; Explain mental model

Retrofitting tests

You fix poorly-written tests and add tests where they're missing.

Excluded: You're only expected to retrofit tests when there's established testing patterns for you to follow.

See also: Sociable unit tests; Narrow integration tests

Exploratory testing

You use exploratory testing to discover your team's blind spots. You start with a charter (a goal for your testing), then identify parts of the system that are related to that goal. You manually exercise those parts of the system, using your experience, intuition, and predefined lists of heuristics about common ways systems fail. For example, you might exercise an image upload feature with a normal image, a zero-pixel image, a million-pixel image, an empty file, and a corrupted file.

You often conduct exploratory testing through the user interface, but you also manipulate databases, modify URLs, look at logs, and so forth. As you work, you take notes about what you learned, both so you can recreate any issues you find, and also to help you identify where to explore next.

See also: Manual validation

Further reading: *Explore It!* by Elisabeth Hendrickson

Advanced Design (Senior SE)

Codebase design

You consider the high-level responsibilities and relationships between namespaces, classes, and modules as you make changes to your team's code. You strive to create structures that maximize cohesion and minimize coupling. You ensure code that's related is close together (cohesion), and that changes to one part of the system don't result in changes to unrelated parts of the system (coupling). You minimize duplication by following the "Once and Only Once" principle: Every important concept to your system has one definitive place where it's represented in code.

See also: Function and variable abstraction; Class abstraction

Simple design

You keep your designs simple, supporting only the features and performance your team requires today. You avoid anticipating future needs or adding “hooks” to make anticipated changes easier. Instead, you write code that is easy to extend in any direction, anticipated or not. You do this by minimizing the amount of code you write, writing good tests, and focusing on cohesion and coupling. When making design decisions, you ask yourself, “When (not if) this decision needs to change, how hard will it be?” and choose the path that minimizes both present and future effort.

In Extreme Programming, this is called “YAGNI” (“You aren’t gonna need it”) and “Do the simplest thing that could possibly work.”

See also: Simple codebase architecture

Reflective design

When you need to make a change, you start by studying the design of the code you’re about to change. You critique its design in the context of the larger system, keeping your new change in mind. You identify design improvements that will both make your change easier and make the system as a whole easier to maintain and modify. You incrementally refactor the code to make those improvements as part of making your change.

You avoid making every design improvement you can think of. You pragmatically balance making the code better, making your change easier, and finishing your work in a timely manner. You always leave the code at least a little bit better than you found it.

See also: Campsite rule

Cross-class refactoring

As you work in the code, you identify ways to improve the responsibilities and relationships between namespaces, classes, and modules. You use cross-class refactorings such as “Move method” to make those improvements. As you refactor, you work in small, behavior-preserving steps, ensuring the tests continue to pass at every step of the way. Each step is only a few minutes of work. You refactor rather than rewriting.

You strike a pragmatic balance between spending time on refactoring and spending time on adding features. You always leave the code at least a little bit better than you found it.

See also: Campsite rule; Codebase design; Method and variable refactoring; Architectural refactoring

Basic database design

You understand the principles of database design, such as primary keys and foreign keys; normalization and denormalization; data types and constraints. You use this knowledge to assist database specialists when your team needs to make changes to its database schema.

Mental model of team dependencies

You have a deep understanding of the key concepts, abstractions, and metaphors required to work with all of your team's complex dependencies. You also have a deep understanding of all dependencies related to your codebase specialty. You're able to come up to speed quickly, with help, when working with dependencies outside your specialty.

See also: Codebase specialty

Evaluate complex dependencies

Given a need for a third-party dependency with a complex API, you evaluate options and choose one that best suits the team's needs. You consider factors such as fitness for purpose, future needs, simplicity, cost, maintainability, support, and longevity. You trade off the costs of using the dependency versus the team building and maintaining something itself. As you consider buy-vs-build tradeoffs, you focus on the cost to build just the parts the team needs, not the cost of replacing the entire dependency.

See also: Evaluate simple dependencies

Simplify and remove dependencies

You evaluate your team's usage of dependencies with a critical eye. You look for opportunities to replace dependencies with code owned by your team, ways to abstract dependencies behind an interface, and opportunities to replace complex or unreliable dependencies with simpler, more reliable options. You discuss those opportunities with your team and manager and come to a joint conclusion about how to simplify.

As a general principle, you recommend removing dependencies when the cost is less than a day of effort, and you recommend keeping dependencies when it's more than a week of effort. However, you always make your recommendation based on engineering tradeoffs such as the dependency's maintenance burden, its reliability, cost for the team to maintain their own solution, and so forth.

Advanced Operations (Senior SE)

Observability

You help your team build software that can be observed in production. When implementing a change, you consider observability needs such as error detection, performance characteristics, intrusion detection and remediation, security and privacy, and business outcomes. You work with stakeholders to find the right balance between observability and speed of development, without sacrificing the team's ability to respond to emergencies. You add logging, monitoring, and alerting to achieve your team's observability goals. You're careful to avoid "log spam" and other noise that makes issues hard to detect and investigate.

See also: Issue investigation

Basic build automation

You have a working knowledge of your team's local build automation (including testing) and are able to modify it to keep up with changes to your team's system. You're also able to troubleshoot problems when the automation fails. Your changes leave the automation at least slightly better than you found it.

Excluded: You're only expected to be able to follow existing patterns in the build automation.

Basic deployment automation

You have a working knowledge of your team's deployment automation and are able to modify it to keep up with changes to your team's system. You're also able to troubleshoot problems when the automation fails. Your changes leave the automation at least slightly better than you found it.

Excluded: You're only expected to be able to follow existing patterns in the deployment automation.

Incident leader

During a production incident, you play the role of incident leader (also known as "incident commander"). You assemble the response team, delegate responsibilities, coordinate efforts, and monitor progress. If additional people or resources are required, you work with management to procure them. After the incident is over, you ensure an incident retrospective is performed, documented, and resulting action items have been assigned.

Incident communicator

During a production incident, you play the role of incident communicator. You maintain awareness of the details of the incident and communicate status to stakeholders. This includes communications to customers, such as updating StatusPage, as well as communications to management and internal stakeholders, such as posting to Slack. You ensure the incident team isn't distracted by stakeholder requests and you ensure communications among members of the incident team are flowing smoothly.

Incident fixer

During a production incident, you play the role of incident fixer. You investigate the cause of the incident, determine how to fix it, and take appropriate action. You communicate effectively with the rest of the incident team, sharing findings and asking other team members to double-check your assumptions. You balance rapid response with careful decision-making. After the immediate issue is resolved, you help the team identify follow up actions to prevent recurrence. You're an active participant in the post-incident retrospective.

See also: Issue investigation

Technical Leads

Technical Leads are the most senior engineers dedicated to a single team. They're leaders, mentors, and advisors. These skill sets represent the foundations required for this pinnacle role.

Team Leadership (Tech Lead)

Personal authority

People on your team look to you as a leader, not because of your position within the organization, but because of the respect they have for you and your skills. Unlike institutional authority, which is based on the power of the organization, your personal authority is granted by the people around you. It can't be forced.

Leadership specialty

You work with managers and senior engineers to define your leadership specialty. Your specialty is a set of technical, leadership, product, and communication skills that you're particularly suited to teaching to others. You tune your specialty to mesh well with the needs of the team and the specialties of other technical leads.

Engineers on the team turn to you for coaching and mentoring within your specialty. You advise managers on the team's capabilities and needs related to your specialty. You work with other technical leads to advise managers on the capabilities and needs of the team as a whole.

See also: Codebase specialty; Mentoring and coaching

Leaderful teams

You work with managers and other technical leads to understand what peer leadership roles your team needs and which team members are suited to filling those roles. You help team members grow into the leadership roles they're suited for. When a situation arises that they have the ability to lead, you encourage them to step forward. For areas where they don't yet have the ability to lead, you provide mentoring and coaching to help them grow their abilities.

Your encouragement is positive and supportive. You push people to excel without putting them on the spot or pressuring them into a situation where they don't feel safe.

Further reading: Leadership Roles

See also: Leaderful team member, Mentoring and coaching

Assess technical skills

As you work with people on your team, particularly in pairing and teaming sessions, you gain a sense of their technical skills. You provide feedback to engineers on how they can improve their skills when asked. You work with managers to help them understand team members' technical skills and progress.

Assess interpersonal skills

As you work with people on your team, particularly in group discussions and teaming sessions, you gain a sense of their interpersonal skills. You provide feedback to engineers on how they can improve their skills when asked. You work with managers to help them understand team members' interpersonal skills and progress.

Assess product skills

As you work with people on your team, particularly group discussions and planning sessions, you gain a sense of their product skills. You provide feedback to engineers on how they can improve their skills when asked. You work with managers to help them understand team members' product skills and progress.

Technical interview

You interview prospective employees or contractors to assess their skills. You follow an interview agenda and rubric provided to you by management. Your interview style is supportive and friendly, looking for opportunities to help the prospect succeed. You're aware of how unconscious bias can affect your judgment and actively take steps to combat it.

Impediment removal

You identify impediments to the team's success. You bring these impediments to the attention of managers and other technical leads, then jointly work to resolve them.

See also: Critique the process, Circles and soup

Interpersonal Leadership (Tech Lead)

Humility

You recognize that others' ideas are as important as your own, and that giving space for others to contribute and take the lead is often more important than taking the lead yourself. When making tradeoff decisions, you consider the effect of the decision on team members' participation and sense of ownership. As a result, you'll sometimes support a less-than-perfect idea contributed by others ahead of a perfect idea contributed by yourself—potentially for use as a teachable moment in the future—although you're careful to balance that against the long term sustainability of your team and software.

You also recognize that your initial judgment about what is and isn't a good idea isn't always correct. You admit errors and celebrate opportunities to be wrong.

See also: Try it their way

Psychological safety

You help encourage a culture of psychological safety on your team. *Psychological safety* is the ability to propose ideas, ask questions, raise concerns, and make mistakes without being punished or humiliated. You help people understand that psychological safety doesn't mean there are no conflicts. Just the opposite: people are safe to disagree with each other, so they do, in a respectful and constructive manner.

You encourage team members to enable each other's voices, be open about mistakes, be curious, give and receive feedback, use empathy, and allow themselves to be vulnerable. You model the behaviors you want to see and are explicit about expectations. You don't shy away from conflict and you encourage constructive disagreement in your team.

See also: Defend a contrary stance

Calm the flames

You're a steady and calming influence in the team, particularly when tempers run high. You don't rush to judgment and you avoid taking sides. Instead, you help people understand each others' points of view. You ensure each person has a chance to speak and explain their point of view. You rephrase and ask for clarification when people are talking past each other.

See also: Try it their way; Active listening

Ignite the spark

You understand what drives the people on your team: what they're passionate about, personally and professionally, and what they get out of being part of the team. You help team members understand how organizational and team goals can help them achieve what they want. When that isn't possible, you help managers understand why, and suggest changes that will help everyone win. You apply the same thinking to yourself: you look for reasons to embrace organizational and team goals, not reject them.

See also: Intrinsic motivation; Growth mindset

Product Leadership (Tech Lead)

Options thinking

In finance, an “option” is the ability to buy something in the future, possibly for less than it would otherwise cost. If you don’t use it, the money spent to buy the option is wasted, but it can still be a good way to decrease risk or create opportunities.

Options *thinking* is about buying something you might not need so you can decrease risk or create opportunities. For example, if you’re concerned that a vendor might raise their prices, you could spend effort building a way to switch to another vendor. If the vendor raises their prices, you switch. If they don’t, the effort was wasted, but it was still worth mitigating the risk.

You incorporate options thinking into your approach to plans and risk management. You treat options as another tool in your toolbox: not something to use in every case, but a powerful way to make your plans more flexible.

See also: Risk management; Comfort with ambiguity

Status and forecasting

You are familiar with the ways status and forecasts are communicated to stakeholders. You ensure the team is aware of their role in updating status and providing estimates as needed. You understand how estimates are converted to forecasts, and you use the team’s forecasting tools to provide new forecasts as needed.

Excluded: You’re only expected to use the forecasting tool. You aren’t expected to modify or maintain it.

Progress and priorities

You have conversations with stakeholders about the team’s progress and priorities. You provide comparative estimates and forecasts when they’re available. You’re calm and collected in the face of stakeholder pressure, focusing on what the team can do with the capacity it has, and identifying ways to cut scope and deliver incrementally when they want faster results. You provide technical justifications for estimates, as needed, without being defensive. You don’t allow yourself to be bullied into providing forecasts or estimates when none exist. You don’t reduce estimates in response to stakeholder pressure when nothing else has changed.

Design Leadership (Tech Lead)

Simple codebase architecture

Codebase architecture is the recurring patterns in your team's code. For example, a web application might give every endpoint a route definition and controller. These patterns lead to consistent code, but they're also a form of duplication, and changing them can be very expensive.

You're reluctant to introduce new architectural patterns, and you guide your team to be conservative in doing so. You introduce just what is needed for the amount of code you have today and the features you support at the moment. Before introducing a new architectural pattern, you ask if there's a need for the duplication it involves. You instead look for ways to replace the duplication with an abstraction, or to allow different parts of the system to use different approaches.

See also: Simple design

Reflective codebase architecture

When working in a part of the code that's outside your specialty, you scan through directories and files to quickly gain an understanding of the architectural patterns in use and how the code fits together. You look past the cruft that's accumulated over the years to understand the original intended beauty in the code's design. You identify how to work within that design to add features and improve crufty code. You critique and look for opportunities to simplify, but error on the side of humility and respect for the existing design. You work within the existing system, not against it. You improve the code with refactorings, not rewrites.

See also: Reflective design

Risk-driven codebase architecture

You identify the architectural risks in your codebase: the features that your codebase can't easily support. For example, a web application that has duplicate currency rendering logic in every controller will have trouble introducing internationalization.

You use your best guess about product direction to identify which risks are the most important to resolve. When deciding how and when to refactor, you prioritize refactorings that lower your biggest risks. (To continue the internationalization example, you might factor out a “Currency” class with a “render” method.) However, you limit your refactorings to improvements that relate to your current work. You don't introduce hooks for features that don't yet exist.

See also: Campsite rule; Simple design

Architectural refactoring

You help your team identify ways to simplify and improve the architectural patterns in its codebases. Before committing to a change, you try it with a real feature the team is implementing, and evaluate how well it works in practice. You give the test implementation some time to sink in—typically, a week or two—before deciding if it's worthwhile. (You undo ideas that don't work out.)

Unless the change is only a few days of work, you help the team implement it gradually, typically limiting refactoring to code they're actively working with. You use techniques such as Architecture Decision Records to provide guidance for the change. You help them remember to continue to introduce the new pattern as they work, sometimes over the course of many months, until the change is complete.

See also: Simple codebase architecture; Method and variable refactoring; Cross-class refactoring

Published API design

You design an API to be consumed by people outside your team. Because published APIs are difficult to change, you're conservative in your approach. You keep the API as small and simple as possible, preferring to add to it in the future over making a mistake today that will be difficult to change. You take extra time to consider your design, including testing it with others, to ensure the API is easy to use, fit for purpose, and is unlikely to need changes other than future additions.

(work in progress; more to be added in the future)