## A computer screen shot of a blue background AI-generated content may be incorrect.

Introduction: Why Roles and KPIs Matter

In a fast-growing startup, people often wear multiple hats. However, as the team approaches 20+ people, **clearly defined job descriptions (JDs)** and **key performance indicators (KPIs)** become crucial. Without clarity on who owns which responsibilities, teams can trip over each other or let critical tasks fall through the cracks. And without metrics, a startup might *feel* busy without actually knowing if it’s succeeding. By defining roles and measuring outcomes, startups ensure everyone pulls in the same direction.

**Job descriptions** set expectations for each role’s mission, scope, and collaboration with others. This alignment prevents both gaps and overlaps in work. Meanwhile, **KPIs** turn gut feelings into concrete data. They reveal whether the team is on track toward its goals or needs to adjust course. In short, JDs tell people *“This is what I own”*, and KPIs answer *“How do we know we’re winning?”*.

Done right, JDs and KPIs together drive focus, accountability, and improvement. They help startup teams make decisions based on facts rather than hunches. According to industry research, setting clear, measurable KPIs provides a “compass” for a startup - guiding every data-driven decision and helping teams pivot or persevere as needed. High-performing tech teams also tend to measure what they do and continuously improve: for example, Google’s DevOps research (DORA) shows that an organization’s software delivery metrics predict its overall performance and even team morale. In practice, this means defining the right metrics (deployment speed, quality, customer happiness, etc.) and tracking them over time. Startups that establish this measurement culture early can spot problems sooner and boost what works.

Finally, clear roles and KPIs support scale. They make it easier to onboard new hires, divide work logically, and maintain quality as the team grows. As one Atlassian team lead noted, getting a diverse team to work together - keeping them motivated, productive, and growing - is challenging but hugely rewarding. It helps when each person knows their purpose and how their performance is measured. What follows is a comprehensive guide to defining key roles common in early-stage startups and a set of KPIs to manage and improve those roles. We’ll also provide checklists, templates, and real examples to make these concepts actionable.

## Checklist: Setting Up Effective JDs and KPIs

Before diving into each role, use this quick checklist to ensure your job descriptions and metrics are set up for success:

* **Identify Core Roles:** List the critical functions in your startup (e.g. Project Management, Business Analysis, Development, QA, etc.). For each, decide on a clear role (or roles) to own that scope. Early on, one person might cover multiple roles, but define the roles distinctly for clarity.
* **Define Each Role’s Mission and Scope:** Write a one-sentence **mission** for each role - what ultimate goal it serves. Outline key responsibilities and boundaries (what the role owns and what it doesn’t). This prevents confusion between, say, a Tech Lead’s duties and a Dev Lead’s.
* **Set “Collaboration Contracts”:** For each role, list its key collaborators (other roles, stakeholders) and what they owe each other. For example, a Business Analyst and Tech Lead might agree that the BA owns *“what needs to be built”* and the Tech Lead owns *“how it will be built,”* and they work together on feasibility trade-offs. These agreements ensure smooth handoffs and no important work is orphaned.
* **Establish 3-5 KPIs per Role:** Identify a focused set of KPIs for each role that reflect its success. Make sure **each KPI aligns with business objectives** (no vanity metrics). Define how to measure it (formula, unit), how often to measure it (cadence), and who is responsible for it.
* **Apply the SMART Test to KPIs:** Every KPI should be Specific, Measurable, Achievable, Relevant, and Time-bound. In other words, tied to a concrete objective, quantified in an understandable way, realistic to attain, aligned with company goals, and reported regularly. If a proposed metric is vague or not actionable, refine it before rollout.
* **Document and Communicate:** Write down the JDs and KPIs in a place everyone can see (e.g. a Confluence page or handbook). Review them in a team meeting to ensure understanding and buy-in. It’s important people know why each KPI matters - e.g. *“We measure deployment frequency to ensure we’re delivering value quickly”*.
* **Avoid KPI Overload:** Track enough metrics to get a balanced view (productivity, quality, satisfaction), but **don’t overdo it**. A few meaningful KPIs beats a dozen that no one remembers. Also, prefer team metrics over individual metrics for collaborative work, to encourage teamwork rather than finger-pointing.
* **Watch for Gaming or Unintended Effects:** Be mindful of Goodhart’s Law (“When a measure becomes a target, it ceases to be a good measure”). For each KPI, think of how someone might game it or narrow their focus excessively, and add safeguards (see “Anti-gaming” in the KPI anatomy below).
* **Iterate and Evolve:** Treat both JDs and KPIs as living documents. Revisit them every few months or whenever the business pivots. Early-stage startups change quickly - roles might need to be adjusted and KPIs refined as you learn more. Continuous improvement isn’t just for code, but for roles and processes too.

## KPI Anatomy - The Makings of a Good Metric

Not all metrics are created equal. A “good” KPI has a clear structure that makes it effective. Every KPI you set should include the following components:

| **KPI Component** | **Description** |
| --- | --- |
| **Name** | Clear, concise title of the metric (e.g. “Deployment Frequency”). Should instantly convey what’s being measured. |
| **Objective** | *Why* this KPI matters - the goal or business outcome it reflects. For example, *“to track how quickly we deliver value to users.”* This ties the metric to a strategic objective. |
| **Definition & Formula** | Exactly what is being measured and how. This includes the unit of measure and formula. (For instance, *“Count of production deployments per week”* or *“Story points delivered / Committed \* 100, per sprint”* for a sprint predictability percentage.) Avoid ambiguity here. |
| **Target or Benchmark** | A quantitative target, range, or industry benchmark to give context. E.g. *“Target 5 deployments/week”* or *“Lead time under 1 day (elite DORA performance)”.* If you have no historical data, you might start with an initial benchmark and refine later. |
| **Cadence** | How often it’s measured and reviewed. For example, daily, per sprint, weekly, or monthly. Consistent reporting intervals make the KPI time-bound and actionable. |
| **Owner** | Who is accountable for this metric. It could be a role (e.g. “TPM is responsible for tracking and reporting this”) or a team. Everyone should know who monitors the number and drives improvement. |
| **Anti-gaming Check** | Potential unintended consequences or ways the metric could be “gamed,” and how to mitigate them. This is a sanity check to ensure the KPI incentivizes the right behavior. (E.g. if measuring lines of code, acknowledge that more code isn’t always good - that’s why we **don’t** use LOC as a KPI.) Ensure each KPI encourages *outcomes* not vanity outputs. |

For example, a poorly defined metric like *“Write more code”* fails many of these criteria - it’s not clearly tied to a positive outcome (more code isn’t inherently better) and would encourage bloated code. A better KPI might be *“Cycle Time from code commit to deploy”* with the objective of faster delivery, measured in hours, tracked per sprint, with engineering team ownership. This is specific, meaningful, and can drive improvement in release processes.

Keep in mind that **no single KPI gives the full picture**. In software teams, you need a balanced set. Nicole Forsgren (co-author of the SPACE framework and DORA research) emphasizes using multiple dimensions of developer productivity - satisfaction, performance, activity, collaboration, and quality - rather than obsessing over one number. The following sections provide role-specific templates, including KPIs, for key startup roles. Each template is designed to cover a broad range of success signals for that role.

Role performance Matrix

Every role is measured against a shared catalogue of performance areas. The accompanying table marks each shared area

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Performance area (normalized) | TPM | BA | Dev Lead | Tech Lead | Dev | QA | TA |
| Agile & Scrum (Lean) | ✓ | ✓ | ✓ |  | ✓ | ✓ | ✓ |
| Communication & Collaboration | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Project execution / Delivery discipline | ✓ |  | ✓ |  | ✓ | ✓ |  |
| Tools / Tooling | ✓ | ✓ | ✓ |  |  | ✓ | ✓ |
| Technology / Technical skills | ✓ | ✓ | ✓ | ✓ |  | ✓ | ✓ |
| Domain expertise / knowledge |  | ✓ |  |  |  | ✓ | ✓ |
| Documentation |  | ✓ |  |  |  | ✓ |  |
| Learning, training & coaching | ✓ | ✓ | ✓ |  | ✓ | ✓ |  |
| Testing mindset & practice (TDD, test cases) |  |  | ✓ |  | ✓ | ✓ |  |
| Leadership | ✓ |  | ✓ | ✓ |  |  |  |
| DevOps & CI/CD practices |  |  | ✓ | ✓ |  |  | ✓ |
| Professional rapport | ✓ | ✓ | ✓ |  |  | ✓ |  |

The actual performance expected under each area may vary by the role involved and the expertise level.

Role Templates

Below are standardized templates for the key roles in an early-stage software startup. Each role section includes: the role’s Mission (its core purpose), the main Responsibilities grouped in logical areas, the role’s Scope Boundaries (what it owns and what it does not), its key Collaboration points with other roles, a set of KPIs (with definitions, measurement cadence, and anti-gaming notes), typical Career Levels and progression signals, common Failure Modes to watch out for, and suggested Weekly Actions for someone in that role. These templates can be adapted to your startup’s specific context. The goal is to make each role crystal clear to the whole team, and to provide a basis for managing and developing the people in those roles over time.

### Technical Program Manager (TPM)

**Mission:** *Ensure the software delivery process runs smoothly and predictably, so the team delivers value to customers on time and continuously improves.* The TPM (sometimes called Technical Project Manager or Scrum Master) is the custodian of the team’s process and execution. They don’t write the code or decide product scope - they empower those who do by removing roadblocks, coordinating work, and using data to drive improvement.

**Key Responsibilities:**

* **Agile Process Facilitation:** Serve as the Scrum Master for the team - facilitate sprint planning, daily stand-ups, sprint reviews, and retrospectives. *Guide the team in Agile/Scrum principles* and coach them towards higher maturity in self-organization. Ensure the *Definition of Done* and other process standards are clear and upheld across the team.
* **Project Execution & Delivery:** Own the delivery timeline of the team’s projects. Track sprint progress and release milestones; proactively identify risks or impediments that could derail commitments. Resolve blockers and conflicts, whether it’s unblocking a developer stuck on a dependency or mediating scope trade-offs with the Product Manager. Keep work moving forward at a sustainable pace.
* **Performance Tracking & Improvement:** Collect and maintain **information radiators** (dashboards, Jira reports) that show the team’s performance data - e.g. sprint burndown, velocity trend, cycle times. Use these metrics to spark discussions in retrospectives and leadership meetings. Drive continuous improvement initiatives (process changes, trainings) based on what the data and team feedback indicate. For example, if velocity is volatile, work on better backlog grooming; if quality issues spike, facilitate a root cause analysis.
* **Stakeholder Communication:** Act as the *main point of contact* for status updates and coordination across departments. Maintain transparency with both internal stakeholders (engineering managers, other teams) and external ones (product owners, possibly clients) regarding sprint status, risks, and mitigations. Escalate issues to management in a timely manner when needed - no surprises. Ensure the Product Manager is kept up-to-date and that the team understands product priorities in turn.
* **Scope & Change Management:** Guard the team’s focus by managing scope changes professionally. Work with the Product Owner (PO) to make sure new requests or backlog changes are properly reviewed and, if accepted, scheduled without jeopardizing current commitments. Protect the team from unrealistic last-minute scope creep, while remaining flexible when truly needed changes arise. This includes saying “not now” or negotiating swapping priorities when appropriate, in order to maintain the integrity of sprint goals.
* **Team Enablement & Coaching:** Practice servant leadership to enable team success. Mentor team members in agile practices, especially if some are new to Scrum. Pair up new hires with buddies, ensure onboarding checklists are followed, and get new team members up to speed on how the team works. Identify skill or knowledge gaps within the team and work with Engineering Management to address them (through training, hiring, or adjusting scope). Foster an environment where team members feel safe raising problems without fear of blame. Essentially, nurture a high-performing, cohesive team culture.

**Other responsibilities:**

* Assess Scrum maturity quarterly and coach the squad toward the next maturity level (backlog health, WIP discipline, DoR/DoD, retro follow-through).
* Demonstrate working knowledge of the 9 PMBOK areas; surface risks and trade-offs across scope, schedule, cost, quality, resources, communications, risk, procurement, and stakeholders.
* Own onboarding to Agile ways of working for new joiners (checklists, buddy assignment, first-sprint goals).
* Maintain information radiators (dashboards, burn charts, blockers board) that reflect current KPIs and decisions.
* Enforce due-date hygiene: set/update due dates within Day-1 of the sprint and keep them accurate through EOD updates.
* Coordinate distributed teams across time zones with explicit SLAs for updates, handoffs, and decision logs.
* Operate Jira/Confluence at an advanced level; standardize boards, fields, and reports for consistent visibility.

**Scope & Boundaries:** The TPM focuses on *how* work gets done rather than the product’s content. They **own the development process and timeline** - ensuring the team’s work is organized and impediment-free. However, a TPM **does not dictate technical decisions** (Tech Leads/Dev Leads do that) and does not set product requirements (that’s the Product Manager or BA). The TPM makes sure that once the *“what”* is defined, the *“how/when”* happens predictably. In terms of authority, a TPM may not have formal managerial power over developers (especially in a Scrum Master model), but they exercise influence through facilitation and data. The TPM should respect boundaries with Engineering Managers (EMs) - e.g. involve the EM if team members are underperforming or there are interpersonal issues outside of process. Also, the TPM is not simply a note-taker or admin; they are a leader in driving continuous improvement, not an order taker from management. Their scope includes process tooling (Jira configuration, etc.) and sometimes program-level coordination if multiple teams are involved in a project.

**Key Collaboration Interfaces:**  
- **Product Owner / Product Manager (PM):** Daily sync as needed to clarify scope and priorities. TPM ensures the dev team gets clear stories; in return, PM relies on TPM for realistic timelines and heads-up on risks. They collaborate during backlog grooming and release planning.  
- **Engineering Manager (EM):** The TPM works with the EM on team health and performance. For example, if the TPM’s metrics show persistent overtime or blockers, the EM and TPM discuss solutions (maybe adjust team capacity or skills). The EM also helps resolve higher-level issues escalated by the TPM (resourcing, cross-team coordination).  
- **Tech Lead / Dev Lead:** TPM coordinates with technical leaders to understand any technical risks or dependencies that could affect scheduling. For instance, the Tech Lead informs TPM of a significant refactor needed; the TPM then plans that into the schedule. They also jointly run retrospectives (tech leads add insight on technical root causes, TPM on process).  
- **Business Analyst (BA):** The BA and TPM ensure that requirements are ready for the team. The BA writes the user stories; the TPM ensures they meet the Definition of Ready (DoR) and are scheduled in sprints. If requirements are unclear, TPM flags it to the BA. In some teams the BA might also fill a Proxy PO role; in such cases TPM works closely to prioritize the backlog.  
- **Quality Assurance (QA) Lead/Engineers:** Work with QAs on testing timelines. E.g., TPM makes sure QAs aren’t overloaded at the end of a sprint by encouraging devs to finish earlier or by staggering story completion. TPM also helps enforce that *“Done” means Done* (including passed QA). If a lot of defects are being found, TPM facilitates a discussion between QA and dev on process improvements.  
- **Other Teams’ TPMs/Leads:** For cross-team projects, TPM coordinates with counterparts on integration points, joint timelines, and dependency management. For example, if Team A’s work depends on Team B’s API, the TPMs from each team sync up regularly to ensure alignment. This often involves shared planning sessions or stand-ups at the program level.

**KPIs for TPM:** *(Measured per two-week sprint, unless otherwise noted)*

* **Sprint Predictability (Commitment Reliability):** *Definition:* The percentage of work planned that is completed within the sprint. Calculated as (Story Points Completed / Story Points Committed) \* 100. *Cadence:* Each sprint. *Target:* ~90% or higher (some carryover is normal, but should be <10%). *Anti-gaming:* Watch out for sandbagging (under-committing too few points just to hit 100%). Emphasize honest planning estimates. The goal is to improve predictability, not to game the ratio by always committing low.
* **Team Velocity Trend:** *Definition:* The total story points completed by the team each sprint, tracked over time. *Cadence:* Each sprint (with a 5-sprint rolling average). *Target:* Stable or gently increasing trend that reflects continuous improvement (e.g. trending from ~20 to ~30 points over a few quarters as the team matures). *Anti-gaming:* Ensure consistency in point estimation. A sudden jump in velocity could mean the team changed point sizing or cut quality corners - investigate abnormal changes rather than celebrating too soon. Velocity is a planning tool, not a score - avoid comparing between teams.
* **Cycle Time (Work Item Lead Time):** *Definition:* The average time from when work on a user story starts to when it is deployed to production (or ready for release). This often includes coding + code review + testing duration. *Cadence:* Measured for each story, reported weekly or per sprint as an average. *Target:* Depends on context; e.g. <5 days for a small feature. *Anti-gaming:* Beware of splitting stories too small just to reduce cycle time, or rushing code through without proper testing. Use this alongside quality metrics (like defect rates) to ensure faster delivery isn’t causing sloppy work.
* **Impediment Turnaround Time:** *Definition:* How quickly the TPM removes reported impediments or blockers. For example, if a developer is blocked waiting for a decision or an environment fix, measure the time from blocker identification to resolution. *Cadence:* Ongoing, log each impediment in an impediment backlog and track time to close. *Target:* 1 business day for critical blockers, 2-3 days for non-critical. *Anti-gaming:* Encourage *raising* impediments early. If no one reports issues, it could be a sign of fear or ignoring problems - not necessarily that the TPM is perfect. So this KPI should be paired with a healthy “impediments raised” count.
* **Team Happiness Index (Qualitative):** *Definition:* A subjective measure of team morale and perceived productivity. Could use a brief quarterly survey (e.g. team rates on a scale 1-5 how happy/productive they feel). *Cadence:* Quarterly. *Target:* Improve or maintain scores (aim for >4.0/5 average). *Anti-gaming:* Absolute honesty is needed - make it anonymous. The TPM shouldn’t chase a number here, but use feedback to foster a better environment. Research shows developers’ satisfaction correlates with higher productivity, so a happy team usually means better output. This KPI ensures the TPM focuses on team health, not just deliverables.

**Career Levels & Growth:** In a startup, a TPM might be the only one, or there could be junior vs senior TPMs as the team grows. Typical progression could be: **Associate Project Coordinator (entry-level)** - assists a more senior TPM with task tracking; **TPM / Scrum Master** - independently handles one team; **Senior TPM** - handles multiple teams or a complex program, mentors other TPMs; **Program Manager** - drives large cross-team initiatives, contributes to process at org level. Growth signals include: the ability to handle larger teams or projects, improving the team’s performance metrics over time, and influence beyond the immediate team (e.g. introducing an Agile practice adopted by other teams). A senior TPM will also exhibit strong strategic insight - not just reporting metrics, but analyzing and acting on them for continuous improvement.

***Signals of excellence:*** The team consistently meets commitments and rarely feels “death march” pressure; stakeholders always know the real status of projects (no surprises); sprint retrospectives yield concrete improvements that the TPM follows up on; team members feel supported and report that the TPM “has everything in hand” process-wise. The TPM is proactive - e.g. they start risk discussions early in planning, and surface process issues before they hurt. They also adapt the process to the team’s needs (not a by-the-book martinet). Perhaps most telling: over a few quarters, the team’s delivery metrics (predictability, cycle time, quality) improve and the team’s morale remains high, indicating the TPM’s positive impact.

*Failure modes:* Common pitfalls include turning into a **“Ticket Admin”** - merely creating Jira tickets and updating boards, without truly leading. Another failure mode is being **overly rigid** with process (enforcing scrum ceremonies or estimates dogmatically even when they don’t add value). Avoid the **velvet rope backlog**, where the TPM becomes a gatekeeper causing delays rather than enabling flow. If a TPM habitually communicates status as “green” until suddenly it’s “red,” that’s a failure in risk management and transparency. Micromanaging the team’s every move (acting like a taskmaster) instead of trusting the team is another anti-pattern. Additionally, if improvement stalls - the team keeps hitting the same retros action items with no change - the TPM may be failing to drive continuous improvement. Lastly, burnout or frustrated team members can signal that a TPM is pushing output at the expense of sustainable pace (e.g. gaming velocity or neglecting the human aspect). A great TPM avoids these traps by focusing on servant leadership, not just the Gantt chart.

**Weekly Action Plan for TPM:**

* **Monday:** Review the new sprint scope with the team. Ensure all stories meet the DoR (clear, estimated, with acceptance criteria). Update the sprint board with any carry-overs. Communicate sprint goals to stakeholders.
* **Daily:** Facilitate the stand-up. Immediately after stand-up, update the Jira board as needed and log any impediments mentioned. Ping owners of blockers opened previously to check status. Communicate any new blockers to relevant stakeholders (e.g. “QA environment down, needs infra help”). Throughout the day, keep an eye on work in progress - gently remind on WIP limits if your team uses Kanban, or ensure stories are not stuck.
* **Wednesday/Midweek:** Do a mid-sprint check-in. Review burndown chart; if it’s way off track, collaborate with the team and PM on a correction (maybe descoping a low-priority item or getting extra help). Also check on upcoming work: is the *next sprint’s* backlog groomed enough? If not, arrange a backlog grooming/refinement session with BA/PO and team.
* **End of Week:** Conduct a brief backlog refinement (if not already done) to line up stories for next sprint - ensure estimates and DoR. Confirm availability for next sprint (any vacations, etc.). If the sprint ends mid-week next week, start preparing the review demo and retro doc.
* **Sprint Review Day:** Coordinate the sprint review meeting. Make sure demo participants are ready. Collect feedback from stakeholders. Immediately after, summarize outcomes for the team.
* **Sprint Retro Day:** Come with data - velocity, any notable cycle times, defect counts. Present these to the team to facilitate discussion on what to improve. Ensure at least 1-2 concrete action items are decided. After retro, document those actions and owners in the retro doc or task tracker. Follow up on previous retro actions (are they done or still pending?).
* **Always:** Keep the **communication channels open**. Post a brief mid-sprint status in Slack or email to stakeholders (e.g. “Sprint 5 is on track, 60% of stories done, no major blockers”). Also, meet one-on-one with the Product Manager or Tech Lead if needed to anticipate any scope or technical concerns. Throughout the week, celebrate small wins with the team - positive reinforcement keeps morale up, which in turn boosts productivity.

### Business Analyst (BA)

**Mission:** *Bridge the gap between business needs and the development team, by translating stakeholder requirements into clear, actionable specifications and ensuring the product built delivers the expected value.* The BA is responsible for understanding what problem we’re solving, detailing the requirements, and constantly aligning the product with user/business expectations. In an early-stage startup, the BA often works closely with product managers or sometimes fills a hybrid PM/BA role, ensuring that the engineering team always has well-defined work in their pipeline.

**Key Responsibilities:**

* **Requirement Gathering & Documentation:** Elicit requirements from stakeholders (internal business teams, product owners, or even customers) and document them thoroughly. This includes writing high-quality user stories and acceptance criteria that meet the *Definition of Ready* so that they pass to development without undue back-and-forth. Maintain a prioritized **product backlog** with enough detail that the team can estimate and start work confidently. Produce necessary artifacts like Business Requirements Documents (BRDs) for larger features when needed, ensuring they cover functional and non-functional needs. Also create process flow diagrams, wireframes, or other visuals to clarify complex requirements - using tools and techniques to convey ideas effectively (e.g. workflow diagrams, UML, mockups). In short, the BA owns the **“what”** of the product at a detailed level.
* **Domain Expertise & Analysis:** Develop deep knowledge of the startup’s domain - whether it’s fintech, e-commerce, healthcare, etc. Understand the business processes, regulations, and market context affecting the product. Anticipate how changes in the business environment (new regulations, competitor moves) might require product changes. The BA should proactively identify gaps or opportunities in requirements - not just gather what stakeholders say, but also *probe and uncover* unstated needs or optimizations. Perform analysis techniques such as stakeholder analysis, cost-benefit analysis, impact analysis for proposed features. Essentially, be the “analytical brain” of the product team, ensuring that decisions are well-informed and that the proposed solutions truly address the problem.
* **Communication & Clarification:** Act as a liaison between the development team and stakeholders. Communicate requirements clearly to developers and QAs - e.g. walk through user stories in planning meetings, answer questions swiftly during development, and ensure no requirement ambiguities remain. Conversely, communicate technical constraints or trade-offs back to business stakeholders in understandable terms. The BA often takes meeting notes in discussions and promptly turns them into action items or updated specs. They must be an excellent *translator*, converting business language to tech and vice versa. When developers raise questions or when testers find an ambiguity, the BA is responsible for getting those answers (either by knowing it or by asking the right stakeholder) and doing so in a timely manner.
* **Agile Participation & Proxy-PO Duties:** Attend all relevant Scrum ceremonies - sprint planning, daily stand-ups (when needed), backlog grooming, sprint reviews, retrospectives. In sprint planning, ensure that the highest priority stories are ready and explain them to the team. During sprints, be available to the team to clarify acceptance criteria or adjust details as needed. If the team lacks a dedicated Product Owner on a day-to-day basis, the BA often fills that gap by prioritizing the dev team’s questions and making minor scope calls (escalating bigger product decisions to the actual Product Owner or product manager). The BA also helps in UAT (User Acceptance Testing) or reviewing the developed features to ensure they meet the intent. In some cases, a BA might even host planning or review meetings if a Product Owner is unavailable (acting as a backup to the Scrum Master/TPM for those specific meetings, though not as a full-time scrum master).
* **Backlog Management & Refinement:** Continuously refine the backlog. Ensure there are always at least ~2 sprints worth of groomed stories ready for development. This includes splitting large epics into manageable stories, adding detail to upcoming stories (criteria, business rules, UX requirements) well in advance, and reprioritizing based on feedback. The BA keeps the **“Ready to Groom”** queue populated and helps facilitate backlog grooming sessions. They also maintain any BA-specific boards or documentation repositories (for instance, a “BA & Lead board” for tracking analysis tasks). Essentially, the BA ensures the development team’s pipeline is always filled with well-prepared work, preventing idle time or rushed last-minute story prep.
* **Post-release Validation & Support:** After features go live, the BA gathers feedback and measures if the business objectives were met. They might coordinate with support teams or directly with users to collect any issues or improvement suggestions. If there are production bugs or change requests, the BA analyzes them to see if they stem from requirement gaps and then updates documentation or backlog items accordingly. Additionally, BAs often assist in preparing *release notes* or training materials for business users when new features roll out (translating the technical changes into business language for external communication). In an early startup, the BA might also support *pre-sales* or demo efforts by explaining how the product meets certain requirements, given their holistic view of the product features. This extends their role beyond just the dev team to the business side, ensuring the delivered software actually solves the intended problem and is adopted successfully.

**Other Responsibilities:**

* Use the agreed tooling stack proficiently (Excel/Power BI, SQL, SharePoint, Confluence/Word) to analyze and communicate requirements.
* Produce the right diagram for the problem (flowchart, DFD/PFD, use-case, ERD, activity, domain model) and keep diagrams versioned with stories.
* Create and curate the product roadmap; align scope with constraints and call out impacts early.
* Report against triple constraint and the 9 PM areas with the PM/TPM; flag scope/schedule/cost variances with options.
* Build simple budgets/forecasts for initiatives and keep them current as scope evolves.
* Support product pricing analyses with clear assumptions and sensitivity checks.
* Communicate roadmap timeline changes at least one sprint in advance to all affected parties.
* Ensure relevant compliance considerations are captured in requirements (e.g., ISO 9001/27001/27701, HIPAA where applicable).
* Post-release, facilitate UAT, record outcomes in Jira, monitor adoption, track KPIs, and drive course corrections.
* Contribute pre-sales artifacts when needed (SOWs, RFPs/RFIs, proposals/PoCs) ensuring traceability to requirements.

**Scope & Boundaries:** The BA’s scope is primarily **requirements and analysis**. They *own the specification of what the development team builds.* However, the BA is **not the decision-maker on product strategy** - that’s usually the Product Manager or founders. The BA gathers input and makes recommendations, but major feature decisions or prioritization calls should be made in conjunction with product leadership. Also, the BA does not manage the developers or the timeline (that’s the TPM/Dev Lead’s job), though the BA must be aware of timeline constraints for phased requirements. The BA should avoid the trap of dictating solutions - instead, they specify *what* is needed and *why*, and let the technical team figure out *how*. For example, a BA would document “System shall allow users to reset their password via email link” (what) rather than deciding “we must use AWS Cognito for this” (how). Boundaries-wise, the BA collaborates with QA on defining acceptance criteria but **does not micromanage testing** - once requirements are handed over, QA/Dev determine how to test and implement, though the BA can assist with test case review for coverage. In summary, the BA’s playground is the problem space and requirements space, bridging to but not encroaching on the solution space owned by engineers.

**Key Collaboration Interfaces:**  
- **Product Manager (PM) / Product Owner:** The BA works very closely with the PM. The PM defines product vision and high-level priorities; the BA fleshes out those ideas into detailed requirements. They likely meet multiple times a week (or daily) to clarify upcoming features, review user feedback, and adjust the backlog. The BA may often be the one to draft initial requirements which the PM then approves. In absence of a formal PM, the BA might take on some product decision-making but should regularly sync with founders or stakeholders for direction.  
- **Engineering Team (Devs & QA):** Day-to-day, the BA is in constant communication with developers and testers. In sprint planning, BA presents each story and its acceptance tests. During development, devs will ask the BA clarifying questions (edge cases, expected behavior for certain inputs, etc.), and the BA must respond quickly to keep work flowing. With QA, the BA ensures test cases align with requirements - for example, reviewing test scenarios to confirm they cover all acceptance criteria and business rules. There’s a lot of back-and-forth: the BA might clarify that a certain scenario is out of scope (so dev/QA won’t waste time on it) or realize a missing scenario and add a story or acceptance test on the fly. This tight collaboration is crucial for building the right thing.  
- **Tech Lead / Architects:** BAs consult with tech leads to understand technical constraints or get effort estimates that inform requirements. For instance, if a BA is drafting a requirement and the Tech Lead mentions a particular approach is technically risky or costly, the BA might work with them to adjust the requirement (maybe a simpler alternative) that still meets business needs. Also, for complex systems, the BA ensures that requirements are feasible and consistent with the system architecture - often needing input from the Tech Lead on what is possible. They maintain a dialogue to avoid writing “fantasy” requirements that can’t be built.  
- **Clients/Users/SMEs:** In some startups, the BA may interface with end users or client representatives (especially if building a B2B product) to gather direct input. They might run requirements workshops, interviews, or surveys. If that’s the case, the BA acts as the voice of the customer within the team. They need to cultivate good relationships with these external stakeholders, speaking their language. Internally, the BA also collaborates with subject-matter experts (SMEs) or department heads relevant to features (e.g. talking to the Head of Sales for CRM feature requirements, or a Compliance Officer for regulatory features).  
- **UX/UI Designers:** If the startup has designers, the BA works with them to incorporate design specs into the requirements. For example, the BA ensures that user stories reference the correct mockups or UX flows provided by design. The BA might help translate design decisions into impact on requirements (e.g. “field X is being removed in the new design, so we must confirm if that data is no longer needed”). In some cases without a dedicated designer, the BA might produce basic wireframes themselves to aid understanding.

- **QA Lead / QA Engineers:** (Yes, they were mentioned as part of Eng Team above, but emphasizing separately.) BAs and QAs can develop a “contract” where the BA provides clear criteria and data conditions, and the QA, in turn, uses them to create test cases. The BA should review any requirements traceability matrix the QA maintains, to ensure every requirement is tested. They also collaborate in UAT: BA may coordinate UAT with business users while QA supports. Essentially, they share the goal that the delivered software meets the business need with high quality. One practical interaction: when QA finds a defect that is actually due to unclear requirements, the BA updates the requirement doc to prevent recurrence, and clarifies it for the fix.

**KPIs for BA:** *(These focus on the quality of requirements and backlog health.)*

* **Requirements Quality (First-Pass Acceptance):** *Definition:* Percentage of user stories that pass their acceptance criteria without requiring rework or causing scope change during the sprint. In other words, stories that were “ready” and did not come back for clarification or extensive changes mid-sprint. *Measurement:* Track each story if during development it had to be significantly rewritten or if it was rejected in testing due to missing requirements. Calculate as (# of stories completed without requirement issues / total stories completed) \* 100 per sprint. *Target:* 90%+. Most stories should flow through without requirement-related hiccups. *Anti-gaming:* Count only true requirement issues (not developer bugs) when calculating. If a dev misunderstood but the spec was clear, that’s not on the BA. However, if many misunderstandings occur, that may indicate specs *weren’t* clear enough. This metric encourages thorough, unambiguous specs, but be careful: it shouldn’t discourage conversation. BAs must still welcome questions - the goal is clarity, not zero questions asked.
* **Backlog Grooming Horizon:** *Definition:* The number of sprints worth of work that is fully groomed and ready for development at any time. E.g. if your team velocity is ~20 points per sprint, do you have ~40 points of ready stories (2 sprints worth)? *Measurement:* Check at sprint start how many upcoming story points (or count of stories) have clear description, acceptance criteria, and estimates. *Target:* At least 2 sprints worth of ready work (or whatever threshold your team agrees on, e.g. always have ~100 points groomed). *Anti-gaming:* Ensure those “ready” stories are truly ready (reviewed by team, estimated, not just marked ready by BA alone). This KPI helps prevent scrambling for work each sprint, but beware of over-stocking the backlog with low-priority items - ready work should also be high priority.
* **Requirement Cycle Time:** *Definition:* Time taken from receiving a requirement (from stakeholder or market need) to having it specified and accepted by the team (ready for dev). For example, if a feature request came in on Oct 1 and by Oct 10 the BA has it broken into user stories ready for sprint planning, that’s a 9-day cycle. *Cadence:* Measure for each major feature or epic, then can average. *Target:* Depends on complexity - perhaps 1-2 weeks for medium features, 1 month for large epics. The aim is to keep analysis moving at a startup pace. *Anti-gaming:* Don’t sacrifice quality for speed - churning out half-baked specs faster isn’t success. This metric should be paired with quality metrics (like first-pass acceptance above). It’s mainly to identify bottlenecks in analysis (e.g. waiting too long for stakeholder input, etc.) and ensure the BA isn’t the team’s throughput constraint.
* **Post-Release Requirement Changes:** *Definition:* Number of significant requirement changes or missed use-cases discovered *after* a feature is delivered, within a certain time (say within 2 sprints of release). Essentially, how often do we realize “We built the wrong thing” or “We need to add X because we forgot during analysis.” *Target:* Zero major missed requirements in critical features, minor adjustments are okay but should be minimal. *Anti-gaming:* This encourages thorough upfront analysis, but one must accept that iterative improvement is normal in startups. So treat this KPI as a retrospective learning tool rather than a strict target - analyze each miss to improve future requirement practices. Also, avoid a blame game; many factors cause changes. But if every delivered feature has scope tweaks immediately, maybe requirements were rushed or stakeholder input was lacking.
* **Stakeholder Satisfaction (Requirements Perspective):** *Definition:* Survey or feedback from key stakeholders (or end users) on how well the delivered features meet their needs and expectations. Could be a simple quarterly survey asking business stakeholders: “On a scale of 1-5, are you satisfied that the dev team is delivering the features you expected?” *Target:* High satisfaction (e.g. >4/5). *Anti-gaming:* Qualitative and prone to factors beyond BA control, so use it judiciously. It’s more of a gut check from the “customers” of the requirements. If low, BA should probe why - was it miscommunication, shifting needs, etc. This keeps the BA aligned with true business value, not just the spec process. It aligns with the idea that focusing on the end user is a strong predictor of success (teams that keep user needs in focus perform better).

**Career Levels & Growth:** A BA role in startups might start as **Associate BA** (entry-level, works on small features or assists a senior BA), then **Business Analyst** (independently handles a product area’s requirements), potentially **Senior BA** (leads analysis for large or multiple projects, mentors others), and **Lead BA/Product Analyst** (strategic, works closely with product leadership on roadmap definition, might oversee other BAs). Growth is evident when a BA moves from just capturing requirements to truly shaping them: e.g. proposing innovative solutions to business problems, or becoming a domain expert who is consulted on business decisions. A senior BA also develops strong facilitation skills - running workshops, prioritization sessions, etc., with senior stakeholders. They might even transition into Product Manager roles as the startup grows, since the skill sets overlap.

*Signals of excellence:* Requirements documents are of consistently high quality - devs and QAs rarely need to guess or assume anything. The BA is praised by developers for providing clarity and by business users for understanding their needs. Over time, the BA’s knowledge of the domain makes them a go-to person for “What’s the impact if we do X?” type questions. They manage changes calmly and keep the backlog reflecting the latest decisions. A top-performing BA also helps improve the process: perhaps they introduce a new user story template that catches non-functional requirements, or implement a backlog review meeting that significantly reduces bugs related to missing requirements. Essentially, things run smoother because of their work. The BA also demonstrates strong empathy - they can see both the user’s and developer’s perspective, which helps them catch issues early (like “developers might misinterpret this, let me rephrase it”). They contribute to team productivity by ensuring minimal wasted effort on wrong or unclear specs.

*Failure modes:* A struggling BA often shows up as poorly defined stories - developers frequently say “these requirements are confusing or incomplete.” If development frequently stalls waiting for BA clarifications, that’s an issue. Another failure mode is becoming a mere scribe - just writing down what others say without analyzing or adding value. In that case, the product may suffer from unchallenged assumptions or missed edge cases. A BA who lacks domain knowledge can become a bottleneck, as they have to go back-and-forth too much; not investing time to learn the domain is a mistake. Also, watch out for scope creep via BA - if the BA keeps introducing new “nice-to-haves” that weren’t agreed on, they might be overstepping or not sticking to MVP mindsets. Being non-communicative is another: if a BA doesn’t proactively update the team when requirements change or doesn’t speak up when a requested feature is very complex, they fail to set the right expectations. And if the startup’s priorities pivot but the BA keeps churning on a now-low-priority spec instead of adjusting, that inflexibility can hurt. Essentially, a BA fails if the team builds the wrong thing or if the team faces avoidable requirement surprises - those are the outcomes to avoid.

**Weekly Action Plan for BA:**

* **Start of Week:** Sync with the Product Manager or stakeholders on any new inputs (new requests, reprioritizations). Review the product roadmap to see if any upcoming feature needs early analysis. Update the requirements status for each ongoing epic (are we awaiting user feedback? pending design? etc.).
* **Backlog Refinement & Grooming:** Schedule at least one backlog refinement session with the dev team this week (if not already done). Prepare by choosing a few top candidate stories for two sprints out, and flesh out details beforehand. During the session, clarify those stories, get estimates or at least initial sizing feedback, and note any questions to follow up.
* **Stakeholder Interviews/Meetings:** If working on a new feature, line up any meetings with stakeholders or users early in the week. For example, meet Sales on Tuesday to understand a new customer requirement, or meet the Compliance officer about upcoming regulation changes. Immediately after each meeting, write up notes and update relevant user stories or create new ones.
* **Mid-week:** Update the *BA & Lead board* (if using one) or backlog with progress on analysis tasks. E.g., move a story from “needs info” to “ready for review” if you got the answer. Check in with developers currently working on features - any questions popped up that you need to answer or research? Provide clarifications promptly (never let a dev block on you for more than a few hours if possible). Also, check test cases written by QA for stories finishing this sprint - ensure they align with acceptance criteria. This could be a quick 30-min sync with the QA to go over any test scenario confusion.
* **End of Week:** Do a **backlog health check**: Is the top of the backlog (for the next sprint) groomed? If not, dedicate time to finish writing those stories or acceptance criteria before the week ends. Touch base with the Product Manager on any changes to priorities that happened during the week and adjust the backlog ordering accordingly.
* **Continuous:** Maintain documentation. If you use Confluence or similar, ensure that any BRD or spec document is updated as things change. For instance, if during development a new valid user scenario was discovered, add it to the documentation for future reference. Likewise, keep the *Definition of Ready* checklist handy and ensure each story slated for next sprint ticks all the boxes (clear story, AC, estimated, dependencies identified, etc.).
* **Friday Demo/Review (if applicable):** In the sprint review, listen carefully to stakeholder feedback on the features delivered. Note any new requirements or changes suggested. Immediately after, discuss with the Product Manager whether these should be logged as new user stories or changes.

### Development Lead (Dev Lead)

**Mission:** *Lead the development team in delivering high-quality software by coordinating technical execution, aligning development work with business priorities, and developing the team’s capabilities.* The Development Lead (also known as Team Lead or Engineering Lead) is a blend of technical contributor and team coordinator. In an early-stage startup, the Dev Lead often acts as the “go-to” person for developers - managing day-to-day development activities, ensuring coding best practices, and making sure the team’s output meets expectations in both timing and quality.

**Key Responsibilities:**

* **Technical Planning & Coordination:** Translate project requirements into actionable development plans. In sprint planning or iteration planning, the Dev Lead helps break down user stories into tasks, estimates effort alongside the team, and ensures that the planned work fits the team’s capacity. They keep an eye on sequencing and dependencies of tasks - e.g. if feature A depends on some backend work first, they ensure that gets scheduled appropriately. The Dev Lead assigns or negotiates task ownership in a fair manner, often considering each developer’s strengths and growth areas. Throughout the sprint, they *monitor progress of development tasks*, adjusting assignments if someone is overloaded or a blocker appears. Essentially, they are responsible for the *day-to-day execution* of development work, making sure everything and everyone is on track.
* **Code Quality & Reviews:** Uphold high coding standards and best practices within the team. The Dev Lead often reviews critical code or at least ensures that a robust peer code review process is in place. They might set up and enforce guidelines for code style, unit testing, commit frequency, and documentation in code. When team members submit pull requests, the Dev Lead either reviews them or assigns appropriate reviewers, and ensures feedback is given promptly. They watch metrics like code review turnaround time - e.g. no PR should linger unreviewed for more than a day. If the Dev Lead spots recurring issues (like frequent bugs due to certain patterns), they address it via team education or updating best practices. Another aspect is maintaining codebase health - the Dev Lead encourages refactoring of messy code, reduction of technical debt, and compliance with architectural guidelines. They might run static analysis tools and ensure the team fixes the highlighted issues. In short, the Dev Lead is the guardian of code quality, balancing speed with maintainability.
* **Mentorship and Team Development:** Provide technical mentorship to developers, especially juniors. The Dev Lead is often the most experienced developer on the team and should actively share knowledge - through pair programming, workshops, or simple availability for questions. They help teammates debug tough problems, giving hints rather than just taking over. When a developer hits a roadblock, the Dev Lead guides them to a solution, building their skills in the process. They also encourage best practices like TDD (Test-Driven Development), encourage writing of sufficient unit/integration tests, and guide design discussions. The Dev Lead can assign challenging tasks to team members as growth opportunities, while ensuring they have support to succeed. Essentially, they elevate the whole team’s technical proficiency over time. A great Dev Lead builds a team such that individuals can eventually handle things without heavy oversight - they grow future tech leaders.
* **Interface with Product/Business:** Act as a liaison between the dev team and product or business stakeholders for technical matters. While the TPM/PM handle project priorities, the Dev Lead often communicates the technical feasibility, estimates, and any technical risks or trade-offs. For example, if a product request is extremely complex to implement, the Dev Lead will explain why and perhaps suggest a simpler alternative. They help ensure that product expectations are in line with what’s technically achievable in a given timeline. In sprint reviews or ad-hoc discussions, the Dev Lead might demo technical aspects or answer stakeholders’ technical questions (e.g. performance concerns, integration impacts). They also ensure that non-functional requirements (performance, security, scalability) that the business might not explicitly state are still considered and either addressed or communicated.
* **Deliverable Oversight & Release Readiness:** Take responsibility for the team’s deliverables being ready for release. The Dev Lead tracks progress against the sprint/release goals, and verifies that acceptance criteria are met for each story from a development perspective. They often do a final sanity check on features - e.g. pulling the latest build and trying the new feature as a user, to catch any obvious issues before it goes to QA or to production. They coordinate with QA to ensure that developers fix critical bugs found in testing within the sprint. If a feature isn’t going to be fully done, the Dev Lead is typically the one to raise that flag early to the TPM/PM and suggest options (like de-scoping or moving it out). They make sure *definition of done* is truly done - including code, tests, documentation, and deployment scripts if any. When it’s release time, the Dev Lead might coordinate the actual deployment in collaboration with DevOps or be hands-on deploying if it’s a small startup. They ensure any release notes or migration steps (e.g. DB migrations) are communicated and executed properly. Essentially, they own the outcome that “the code we wrote this iteration is successfully delivered to users.”

**Scope & Boundaries:** The Dev Lead is a **team-level leadership** role. They usually have a foot in both technical work (they often code on critical parts or prototypes) and coordination. Their scope covers the execution of development - they make day-to-day decisions like “we will refactor module X this sprint to simplify adding feature Y” or “Developer A will pair with Developer B on this complex task.” However, the Dev Lead is typically **not a formal people manager** (though in some startups they might be if no Engineering Manager exists). This means they should mentor and guide, but performance evaluations or HR matters are either handled by an EM or co-founders. The Dev Lead should not unilaterally change project priorities or deadlines - they feed input to PM/TPM but respect the process for scope/priorities. Another boundary: the Dev Lead should refrain from becoming a bottleneck - e.g. **not** insist on reviewing every single code change if it slows things down; instead, they should enable team members to review each other. They also need to delegate and trust others rather than micro-manage every technical detail. In terms of technical vs product decisions, the Dev Lead holds the reins on **how** to implement (architecture of the solution in detail), but if a trade-off impacts user experience or scope, they involve the PM/BA to decide.

**Key Collaboration Interfaces:**  
- **Technical Program Manager (TPM)/Scrum Master:** The Dev Lead works closely with the TPM in planning and executing sprints. The TPM handles process and facilitation, while the Dev Lead provides the technical perspective - e.g. how much the team can bite off, whether a certain sequence is efficient. During daily stand-ups, the Dev Lead might help answer TPM’s questions about why something is blocked or provide an update on technical impediments that the TPM can then help remove. They coordinate on timelines: if the Dev Lead sees a risk of slipping due to technical hurdles, they inform the TPM early. In retrospectives, the Dev Lead and TPM might jointly address issues (process vs technical). This partnership is crucial; one can think of TPM as driving *“are we doing things right process-wise?”* and Dev Lead as *“are we doing things right technically?”*.  
- **Product Manager / Business Analyst:** For requirement clarifications and technical feasibility discussions, the Dev Lead often interfaces with the PM or BA. When a user story isn’t clear or seems to conflict with a technical constraint, the Dev Lead will discuss it with the BA to find a resolution (maybe the requirement can be adjusted or an alternate approach can meet it). The Dev Lead also provides effort estimates to the PM/BA for roadmap planning. In some cases, they may suggest technical enablers that should be added to the backlog (like *“We need to upgrade to a new API before we can add these features - let’s plan that work”*). They need to communicate in a way the PM/BA understands - bridging tech and business. For example, explaining that implementing a seemingly small feature will actually require an extensive refactor due to legacy code, thus negotiating either more time or a phased approach with the PM. The Atlassian team lead anecdote illustrates this balance: the Dev Lead often feels pulled by product wants, design ideals, quality needs, and dev reality all at once, and they must find a workable middle ground.  
- **Tech Lead / Architects:** Sometimes a team has both a Dev Lead (team/process-focused) and a Tech Lead (architecture-focused). If so, the Dev Lead collaborates with the Tech Lead on design decisions and technical direction. The Tech Lead might define the overall architecture or technology choices, while the Dev Lead ensures the team follows that vision in daily work. In our context, if “Tech Lead” is a separate role (see next role section), then the Dev Lead focuses more on people/project coordination and the Tech Lead on deep technical design. They should operate as a duo: e.g. when planning a new feature, the Tech Lead outlines how the system should be structured, and the Dev Lead figures out who will do what and how to schedule the work. In smaller teams where one person plays both roles, this distinction is moot.

- **Developers (Team Members):** Obviously, the Dev Lead’s primary collaboration is with the developers on their team. They should foster an environment of open communication, where developers feel comfortable raising technical concerns or new ideas. The Dev Lead might run a weekly dev huddle or design review meeting where any team member can propose improvements. They pair program or mob program with the team on tricky sections. It’s important the Dev Lead doesn’t become “above” the team - they are still a team member and should cultivate trust. When a developer hits an obstacle, they go to the Dev Lead for help; when they finish work early, the Dev Lead might suggest the next priority task or ask them to help a teammate. The collaboration here is part leadership, part service - the Dev Lead leads by serving the team’s needs to excel.  
- **Quality Assurance (QA) / Test Automation Engineers:** A Dev Lead works with QA to ensure quality is baked in, not just tested at the end. For instance, they make sure developers are providing testable builds, that they fix bugs promptly, and perhaps even help prioritize bug fixes vs new features. If QA indicates a lot of bugs in a particular area, the Dev Lead coordinates a mini-hardening or refactoring in that area. The Dev Lead might also review automated test coverage results with the QA automation engineer to identify gaps. They ensure that dev and QA have a good working relationship (e.g. encourage devs to sit with QA to reproduce a tricky bug, etc.). They also enforce the rule that no feature is done until QA signs off - supporting QA’s authority in the process. In collaboration terms, the Dev Lead may convene a quick triage with QA whenever a new bug is reported: assess severity, assign it, and decide whether it blocks the release. This close loop ensures quality isn’t sacrificed for speed.  
- **Operations/DevOps (if separate):** If the startup has a separate Ops or DevOps role (or uses infrastructure services), the Dev Lead coordinates on deployment, environment, and performance matters. For example, ensuring the infra is ready for a new module the team built, or that appropriate logging/monitoring is added. The Dev Lead communicates requirements like *“We need a staging environment with X configuration by next week for testing a new integration”* to Ops. Also, if production incidents occur, the Dev Lead likely leads the technical investigation and works with Ops to pinpoint the issue (might participate in post-mortems and drive follow-up actions such as optimizations or adding alerts). If no separate Ops, the Dev Lead might be the one setting up CI/CD pipelines and ensuring the team’s code gets smoothly from repo to production.

**KPIs for Dev Lead:** *(These measure team output and code health, which the Dev Lead influences.)*

* **Sprint Delivery Rate:** *Definition:* Percentage of committed stories (or story points) delivered by the end of the sprint by the dev team. (This is similar to the TPM’s sprint predictability metric, and indeed is often a shared metric.) *Target:* ~90% completion of planned work consistently. *Anti-gaming:* The Dev Lead should guard against pushing developers to over-commit or under-commit just to hit a percentage. The goal is realistic planning and steady execution, not playing to the metric. As a Dev Lead, if this rate is low or unpredictable, it’s a signal to improve planning or remove blockers. It’s a direct outcome metric of how well the dev team (led by the Dev Lead) is executing.
* **Code Review Turnaround Time:** *Definition:* Average time a pull request stays open before being reviewed and merged. *Target:* e.g. < 1 day for review to start, < 2 days to merge after submission (for normal-sized PRs). *Anti-gaming:* Don’t let developers game this by merging without review. The intent is to ensure prompt feedback and integration, so measure only legitimate reviewed PRs. If turnaround is slow, the Dev Lead should facilitate - maybe assign reviewers more clearly or set aside review times each day. Quick code reviews prevent bottlenecks and context switching overhead.
* **Deployment Frequency (Team Level):** *Definition:* How often the team successfully releases software to production (or to an end-user environment). This is a DORA metric indicating throughput. *Target:* Depending on your startup, could be anything from multiple times per day to every two weeks. Generally, increasing frequency (without sacrificing quality) is good - it means smaller, incremental releases. *Anti-gaming:* The Dev Lead should ensure that frequent deployments are meaningful (not deploying trivial changes just to bump numbers) and stable. If deploying more often, watch the **change failure rate** too. This metric encourages the Dev Lead to automate and streamline the pipeline - if the team can only deploy monthly due to manual processes, improving that is a ( leadership move.
* **Escaped Defect Rate:** *Definition:* Number of bugs that escape into production per release (or per quarter), i.e. customer-reported issues that were not caught in development. *Target:* Ideally zero high-severity issues; for less critical bugs, track count with a trend toward reduction. *Anti-gaming:* Encourage a quality culture rather than hiding bugs. If a bug is found in prod, log it. This KPI pushes the Dev Lead to strengthen code reviews, testing, and requirements to catch issues earlier. But be careful - not all bugs are equal, so focus on significant ones. The Dev Lead can work with QA on root cause analysis for each escaped defect (was it a requirements miss, a lack of test, a rush job?) and then address the cause. This metric ties to the Dev Lead’s ability to ensure quality in the dev process.
* **Team Technical Skill Growth (qualitative):** *Definition:* A qualitative measure, perhaps via 360-feedback or self-assessment, of how team members feel their technical skills are improving under the Dev Lead’s guidance. For instance, a quarterly pulse where developers answer if they have learned new skills or improved in the last quarter. *Target:* High agreement (e.g. 80%+ feel they are learning). *Anti-gaming:* It’s subjective, but it reminds the Dev Lead to invest in people, not just product. If this is low, it could mean the Dev Lead isn’t mentoring or the work isn’t challenging enough. The Dev Lead can then take action like pairing juniors with seniors, introducing tech talks, etc. While not a typical “KPI”, it aligns with the role’s mission to build capability. And a more skilled team will yield better throughput and quality long-term - which connects to the SPACE framework’s emphasis on satisfaction and learning alongside performance.

**Career Levels & Growth:** In startups, the **Dev Lead** role might be the first step into leadership for a developer. Growth could be: **Senior Developer** → **Development Lead/Team Lead** (leads one team) → **Engineering Manager or Architect** as the company grows. Some Dev Leads remain as high-level technical leaders (moving toward Staff Engineer or Architect roles focusing on tech vision), while others transition to **Engineering Management** focusing on people and multiple teams. As a Dev Lead progresses, they might take on leading multiple teams or a whole module of the product, becoming more of a **Tech Lead Manager** (both managing and technically guiding). Signals of seniority include: the ability to improve engineering processes at team or org level (not just within their own team), successful delivery of complex projects spanning multiple sprints or teams, and coaching other leads. A seasoned Dev Lead also starts contributing to hiring (evaluating candidates) and resource planning with higher management.

*Signals of excellence:* The team consistently delivers releases on time with high quality - this implies the Dev Lead is effectively coordinating and guiding the team. The codebase under this lead is clean, maintainable, and has few critical bugs, indicating good technical stewardship. Developers on the team get better over time - you see juniors becoming solid independents, intermediates taking on more complex tasks - showing the Dev Lead’s mentorship impact. Moreover, the Dev Lead fosters innovation: perhaps team members experiment with new tools or improve performance significantly under their leadership. They are proactive in solving problems: if something in the development process isn’t working (e.g. flaky tests or slow builds), the Dev Lead rallies the team to fix it rather than tolerating it. The best Dev Leads also have a *happy team* - while they push for results, they maintain morale, preventing burnout and ensuring recognition for good work. A telling sign: when asked, the team says they feel confident about their work and supported by their lead. Also, stakeholders trust the Dev Lead - if they say a feature will be ready next week, it happens; if an issue arises, they communicate it early and honestly. Essentially, reliability, quality, and trust mark the excellent Dev Lead.

*Failure modes:* A Dev Lead can fail in several ways. **Micromanagement** is one - not trusting team members and trying to control every commit or decision, which can demotivate developers and slow down progress. The opposite, **abdication**, is another - being too hands-off, leading to lack of coordination, inconsistent code, and missed deadlines (the team is drifting without a rudder). A common pitfall is continuing to behave like an individual contributor and not allocating time to lead - e.g. the Dev Lead focuses only on their own coding tasks and neglects to review others’ work or help others, so issues compound. **Poor communication** is a serious failure: if the Dev Lead doesn’t surface delays or blockers to the PM/TPM, the project can derail; similarly, if they don’t keep the team informed of changes or don’t solicit input, it can breed confusion or resentment. Another failure mode: **favoritism or uneven delegation** - e.g. always giving challenging tasks to the same senior dev and trivial ones to juniors, limiting team growth and causing bottlenecks. Also, being **resistant to feedback** - if team members or QAs bring up quality issues and the Dev Lead dismisses them to “just ship it,” technical debt might balloon and people lose trust. Lastly, **burnout of the team** is a red flag: if the Dev Lead constantly overcommits the team or pushes them to work unrealistic hours to meet deadlines, it might achieve short-term results but at the cost of team sustainability (and likely quality issues). In sum, if product delivery is consistently missing expectations, quality is dropping, or team members are unhappy under a Dev Lead, these are failures that need addressing. A self-aware Dev Lead should seek feedback regularly to avoid these.

**Weekly Action Plan for Dev Lead:**

* **Sprint Planning Day:** Before planning, review the upcoming stories thoroughly. Ensure you understand the technical implementation of each and identify any prerequisites (e.g., “we need to create a feature flag first” or “will need an API from another team”). During planning, work with the team to break stories into tasks. Volunteer technical insights like “We can reuse the payments module for this, so task X will be smaller.” Make sure no developer is over-tasked or under-tasked - adjust assignments collaboratively. After planning, communicate the sprint’s technical game plan: e.g. “Alice will focus on backend, Bob on frontend for Feature A, and we’ll pair on the integration mid-sprint.”
* **Daily Stand-ups:** Come prepared knowing the status (via Jira or chats) so you can spot if someone is stuck or if a task is taking longer than expected. If a blocker is mentioned, ensure it’s being handled (TPM may drive removal, but you might need to, say, help with a technical workaround). Offer help: “Do you need a hand with that bug?” or “Let’s sync after this to debug that issue.” Take note of any unexpected work or scope creep popping up - you may need to raise to the PM if it threatens sprint goals.
* **Mid-week Technical Sync:** Host a short mid-sprint check-in (outside of stand-up) to discuss technical issues in depth. This could be a quick team huddle: *“How is the new module design coming along? Any concerns?”* Or discuss any refactoring that might be needed to stay on track. Use this time to review any key pull requests together if needed (maybe a group code review on a tricky piece of code). Ensure that code reviews are happening promptly - if you see PRs aging in the repo, assign reviewers or do it yourself.
* **Mentor Moments:** Each week, schedule a 1:1 or casual check-in with one of your developers (especially juniors). Ask how things are going, what they’re finding challenging, and offer feedback or tips on their work. For example, if a junior dev delivered a feature but the code had some issues, have a constructive chat: *“I liked your approach on X, let’s talk about how to handle Y better - maybe using a design pattern Z.”* Over time, rotate through team members. This investment in mentorship pays off in team improvement.
* **End of Week Progress Review:** By Thursday/Friday, identify what may not get done by sprint’s end. If a feature is lagging, decide in coordination with the team and TPM whether to trim scope or carry it over. Communicate any likely spillovers early to the PM. Also review the quality: run the test suite, see if any new high-severity bugs appeared. If bugs are piling up, possibly allocate last day of sprint as a “bug bash” or fix day. Basically, do a mini “release readiness” check: *Are we on track to deliver what we promised?* If not, act (reallocate help to a task, or inform stakeholders).
* **Release & Retrospective:** If a release is going out this week, oversee the deployment (or assist whoever is deploying). Make sure developers are on standby in case of hotfixes needed. After deployment, congratulate the team on successes (celebrate a bit!). In the retrospective, be open about what went well or not from a dev perspective. If for example code reviews were a bottleneck, suggest a solution (like “let’s try a buddy system for faster reviews”). Encourage team members to speak up about any pain points. Use the retro outcomes to plan a small process improvement in the next sprint - showing that you take action on issues builds team trust.
* **Always:** Keep an eye on technical debt. Each week, if something is repeatedly causing minor headaches (e.g., “module X is so fragile, it broke again”), log it and discuss when you can improve it. Perhaps add a refactoring story to the backlog or bring it up in planning. A bit of continuous maintenance will prevent bigger issues later. Also, stay updated on the broader codebase: skim through others’ commits even if you’re not the reviewer, just to stay in touch with all parts of the project. And maintain documentation - if the team has internal docs or a README, update it when you change something significant. It may seem like extra work now, but it greatly helps onboarding and reducing future confusion.

### Technical Lead (Tech Lead)

**Mission:** *Provide technical vision and ensure the architectural integrity of the product, while guiding the engineering team in solving hard technical problems.* The Tech Lead (or Technical Architect, depending on the organization) is focused on the *how* at a deep level - designing the software architecture, making key technology decisions, and elevating the technical bar of the team. In an early-stage startup, a Tech Lead might still code a lot, but their unique value is in seeing the big picture of the system’s design and steering the team accordingly.

**Key Responsibilities:**

* **System Architecture & Design:** Own the high-level architecture of the application or system. The Tech Lead decides or guides decisions on how the different components of the system interact (e.g. client-server structure, microservices or monolith, database choices, integration patterns). When new features are proposed, the Tech Lead designs technical solutions that satisfy the requirements and fit into the existing architecture without breaking it. They create design documents or diagrams for major changes, evaluating different approaches and discussing trade-offs (e.g. “Should we use an event-driven approach for this, or a direct API call? Here are the pros/cons.”). They also ensure key **quality attributes** (scalability, performance, security, maintainability) are considered in every design[[1]](https://www.geeksforgeeks.org/hr/team-lead-vs-tech-lead-difference-between-team-lead-and-tech-lead/#:~:text=Functionalities%20of%20a%20Tech%20Lead,can%20include)[[2]](https://www.geeksforgeeks.org/hr/team-lead-vs-tech-lead-difference-between-team-lead-and-tech-lead/#:~:text=,arise%20during%20the%20development%20process). A Tech Lead often initiates and leads architecture review meetings for big decisions, incorporating input from other senior devs, and finalizing the design. In short, they are responsible for the *technical blueprint* of the product.
* **Technology Stack Decisions:** Stay on top of relevant technologies and decide when the team should adopt new tools or frameworks. For instance, choosing the web framework, deciding on a testing library, or evaluating cloud services to use. The Tech Lead does due diligence - prototypes or researches - before introducing a major tech change. They also set standards for using the tech stack effectively (e.g. coding conventions, use of certain libraries, recommended patterns). If the startup needs to integrate a third-party service (say for payments or messaging), the Tech Lead assesses the technical feasibility, suggests which provider to use, and outlines how to integrate it. They balance innovation with stability - not adopting every shiny new thing, but also not letting the stack become outdated. Part of this is also **technical risk management**: identifying risky areas (like “our current database might not scale beyond 10k users”) and planning mitigations or upgrades proactively.
* **Code Reviews & Quality Oversight:** While the Dev Lead might focus on day-to-day code reviews, the Tech Lead looks at code with an eye for architectural consistency and long-term maintenance. They review critical or complex code (especially around core architecture or algorithms) to ensure it aligns with the intended design and doesn’t introduce problems. The Tech Lead might establish a “definition of done” for code quality - e.g. mandates for unit test coverage, performance benchmarks for certain modules, etc. They often handle the most challenging code reviews, such as those involving concurrency, security, or significant architectural impact. Additionally, they keep an eye on code metrics like coupling, cyclomatic complexity, etc., often via static analysis tools or architecture fitness functions. If something in the codebase is growing messy or inconsistent with the architecture, the Tech Lead flags it and works with the team to refactor or address it. In essence, they act as the **software custodian**, making sure the code evolves cleanly and according to plan.
* **Mentoring & Technical Leadership:** Guide engineers in solving complex technical issues. The Tech Lead is the one people go to when a bug is extremely hard to trace or a piece of logic is very tricky. They won’t always solve it for you, but they will provide insights, approaches, or pair-program on the toughest bits. They also mentor developers in improving their design thinking - for example, a developer might draft a design for a new feature and bring it to the Tech Lead for feedback, and the Tech Lead will critique and suggest improvements (like “this module could be more decoupled if we apply X pattern”). They may run internal tech talks or training sessions to spread knowledge on areas like “design patterns”, “writing efficient SQL”, or new technologies the startup is exploring. Part of their leadership is also **code by example** - they likely handle some of the hardest tasks themselves, setting a high standard in the code they write for others to learn from. However, a good Tech Lead also knows when to step back and let others implement under guidance, to grow the team’s abilities.
* **Scalability and Performance Tuning:** Proactively plan for and address scaling concerns as the product and userbase grow. The Tech Lead monitors system performance (e.g. with metrics on response times, database load, etc.) and identifies bottlenecks. They design solutions to improve performance, like implementing caching, optimizing queries, or refactoring heavy algorithms. Before scaling pain becomes acute, the Tech Lead often conducts capacity planning: *“If we get 10x users, what component will break first? Let’s shore that up now.”* They also ensure the system is designed to handle expected growth, often advocating for small refactors or tech debt paydown in each sprint that will keep the system robust. In a way, the Tech Lead thinks a few steps ahead technically, so the startup isn’t caught flat-footed by success. Security is another aspect - while maybe a security specialist would be ideal, often the Tech Lead ensures basic security practices (encryption, safe storage of credentials, input validation) are in place and potentially engages external audits when needed.

**Scope & Boundaries:** The Tech Lead’s domain is **technical decisions and design**. They typically do not have direct authority over timelines or task assignment (that’s Dev Lead/TPM/EM). However, a Tech Lead can and should influence those by making sure technical tasks (refactoring, infrastructure work) are represented. They should be careful not to become a bottleneck or single point of failure for decisions - while they are the chief architect, it’s healthy to delegate smaller design decisions to team members and only enforce critical patterns. Also, the Tech Lead is not the same as a personal “code police” - they set standards and review, but shouldn’t block progress over minor stylistic issues; they need to gauge what’s architecturally important vs personal preference. Another boundary: they should collaborate and convince rather than dictate. A Tech Lead must justify their decisions to the team and often to non-technical stakeholders in terms of business impact (“We need to refactor this module for scalability, *here’s why it matters to our customers/performance…*”). If a Tech Lead decides everything in a vacuum, that can alienate the team. So they operate with influence and reasoning. In early startups, the Tech Lead might also be writing a lot of code - but they must balance that with their design oversight; if they focus only on their own code and ignore others’, architecture can diverge. So a boundary is managing their own coding time vs guiding others - they should not take on so much coding that they can’t fulfill their lead duties. Finally, the Tech Lead does not manage people’s careers or performance formally (unless they also wear an EM hat), though they provide feedback; they should be mindful to stick to technical feedback and let management handle HR matters.

**Key Collaboration Interfaces:**  
- **Development Lead / Team Lead:** As mentioned earlier, if both roles exist, the Tech Lead works hand-in-hand with the Dev Lead. The Tech Lead provides the architectural direction, the Dev Lead ensures day-to-day development aligns with that direction. They likely consult each other multiple times a week. For example, the Dev Lead might say “We’re having trouble implementing the design for X within the sprint” and the Tech Lead might adjust the design or suggest a phased approach. Or the Tech Lead might notice in a code review that a solution deviated from the intended design and discuss with the Dev Lead how to correct it (maybe the Dev Lead missed it or there was time pressure). Essentially, the Tech Lead relies on the Dev Lead to carry out the plan within each sprint, and the Dev Lead relies on the Tech Lead to provide a solid plan and help with big technical hurdles. They should present a united front: in planning, if the Tech Lead says “we need to do refactor Y,” the Dev Lead helps schedule it; in execution, if the Dev Lead says “we can’t finish feature Z without cutting corners,” the Tech Lead helps find a better solution rather than forcing a bad implementation.

- **Engineers/Developers:** All developers should feel they can approach the Tech Lead for technical guidance. The Tech Lead might do design sessions with the whole team for new major features - brainstorming approaches and then converging on the best one. They encourage developers to propose ideas too. In code reviews, the Tech Lead provides constructive criticism and explanations, effectively upskilling the team. They also solicit feedback - e.g., after a big release, they might ask the team “What gave us the most trouble technically? How can we improve it?” thereby involving them in the next steps. If a developer strongly feels a certain tech decision is wrong, the Tech Lead should hear them out and reconsider or explain the rationale thoroughly. Collaboration means the Tech Lead isn’t an ivory-tower architect; they sometimes pair with devs on complex tasks and often implement core frameworks that others then use. The team’s buy-in to the Tech Lead’s architecture is crucial, and that comes from involving them and being open to practical feedback (like if something is too complex to implement under startup constraints, maybe a simpler design is warranted).  
- **Product/Business Stakeholders:** While mostly the Dev Lead/TPM interface with product on scheduling, the Tech Lead occasionally needs to engage product folks for *technical discovery*. For example, if the company is considering a big new capability (say, real-time analytics features), the Tech Lead might join early discussions to gauge complexity and perhaps influence scope (“If we limit feature X, we can build this in 1 month instead of 3”). They might create technical presentations or proof-of-concepts to show the PM or leadership what’s possible and how it would work. Additionally, when there are outages or performance problems affecting users, the Tech Lead often has to explain to non-tech stakeholders what went wrong and what’s being done about it (in lay terms). So they collaborate by providing transparency and education on technical risks or needs - for instance, convincing the business why some time must be allocated to scale the system. Good Tech Leads speak the language of business value when advocating technical work.  
- **DevOps/Infrastructure:** If the startup has DevOps engineers or uses cloud services, the Tech Lead works closely with them on how the software is deployed and run. For example, designing a CI/CD pipeline, choosing containerization vs direct deploy, setting up auto-scaling - the Tech Lead will specify requirements and often co-develop these solutions with whoever manages infra. They ensure that architectural decisions align with infra capabilities (no point designing something that can’t be easily hosted on your current platform). In absence of dedicated ops, the Tech Lead might themselves be the one to configure AWS, Docker, etc. but ideally they’ll involve other team members to share knowledge. They also collaborate on monitoring - deciding what to log, what metrics to track for system health, what alerting thresholds to set, etc., often with input from ops or SREs if available.

- **External Technical Contacts:** This can include technical contacts at partner companies or vendors. For instance, if integrating an external API, the Tech Lead might discuss with that partner’s engineer to clarify usage or performance limits. Or if using an open-source framework heavily, the Tech Lead might engage with that community (filing issues, maybe contributing fixes or staying updated on changes). Essentially, they represent the technical face of the startup externally when needed. If the startup’s product has a technical integration with a client’s system, the Tech Lead might be involved in those conversations to ensure both sides connect systems correctly. Collaborating externally helps the Tech Lead foresee integration challenges and plan around them.

**KPIs for Tech Lead:** *(These focus on system-level quality and technical throughput.)*

* **System Uptime/Stability:** *Definition:* Percentage of time the application is available and functioning correctly (could be measured via uptime monitoring or error rate thresholds). Also possibly count of major incidents or outages per quarter. *Target:* For a production user-facing system, typically >99.5% uptime (downtime mainly for planned maintenance if any). The Tech Lead isn’t solely responsible for uptime, but as the architecture owner they strongly influence it. *Anti-gaming:* Ensure you’re measuring real user-impacting downtime. If something is up but in degraded state, count that appropriately. The Tech Lead should use this KPI to drive reliability improvements (adding redundancy, better error handling). It aligns with DORA’s focus on stability metrics like MTTR (mean time to recovery) - a Tech Lead should care not just how to build features, but how to keep them running.
* **Performance Metrics (Latency/Throughput):** *Definition:* Key performance indicators for the system’s speed or capacity. For example, API response time (p95 latency), or the system’s throughput (requests per second handled), or page load time for the web app. *Target:* Specific to the product (e.g. “p95 API latency < 200ms under normal load”). Targets likely come from product requirements (like “page must load in under 3s on average”). *Anti-gaming:* Don’t optimize just the benchmark while neglecting real user scenarios. Use this KPI to ensure the architecture is efficient. If these metrics regress (as more features add bloat), it’s on the Tech Lead to initiate optimizations or architectural changes. Improving these metrics adds directly to user satisfaction and scalability.
* **Technical Debt Burndown:** *Definition:* A qualitative or quantitative measure of technical debt being addressed. Could be number of high-priority refactors or architectural improvements completed in a quarter, or a debt index (like a static code analysis score) improving. Some teams simply track a “tech debt backlog” and measure how many items are closed vs opened. *Target:* Steady reduction of critical debt items over time (or at least keeping the debt backlog from growing unabated). *Anti-gaming:* Make sure what you label as “tech debt” truly is that and not just any task. The Tech Lead should champion this KPI by scheduling regular debt payment. If new debt is incurred (rushed implementations), they ensure it’s flagged to fix later. The KPI is there to prevent the architecture from decaying. For example, if a quick hack was done to meet a deadline, the Tech Lead makes sure a ticket exists to revisit it. Monitoring the ratio of internal improvements to feature work can be useful (maybe aim for e.g. 15% of each sprint on internal improvements). This shouldn’t override delivering features, but it’s a balancing indicator.
* **Lead Time for Changes (Engineering Lead Time):** *Definition:* The time from a feature being defined (ready for dev) to the feature being live in production. This is another DORA metric, reflecting how quickly the engineering system can deliver value. While it depends on many factors (team, process, etc.), the Tech Lead influences it by promoting efficient development practices, continuous integration, and removing architectural blockers to fast delivery. *Target:* As low as possible without quality loss - e.g. if currently it takes 4 weeks to get a minor change out, try to reduce to 2 weeks or less. *Anti-gaming:* Don’t cut necessary testing or quality checks just to go faster - that could increase failures. Instead, the Tech Lead uses this KPI to push automation (CI/CD), modular design (so small changes can be deployed without full regression), etc. It correlates with agility; faster lead time means architecture and processes are streamlined. If it’s very slow, the Tech Lead should find out why (maybe environment setups are manual - then automate them, etc.).
* **Innovation Metric (technology improvements):** *Definition:* A fuzzy metric to track how the product’s technology is staying modern or improving. For instance, number of significant tech upgrades done (upgrading major library version, adopting a useful new tool) in the last 6 months. Or a “innovation score” where you subjectively rate whether the codebase is using up-to-date approaches. *Target:* Continuous improvement - e.g. at least one notable upgrade or new technology adoption per quarter, driven by clear benefit. *Anti-gaming:* It’s not about using new tech for the sake of it. The Tech Lead should justify each upgrade as improving security, performance, developer productivity, etc. This metric simply prevents stagnation. In startups, you can’t afford to rebuild from scratch every year, but you also don’t want to be stuck on outdated tech that slows development or hiring. The Tech Lead can use this to advocate, say, “We should upgrade our framework before it becomes a liability” or “Let’s invest time in switching to infrastructure as code for deployments,” etc.

**Career Levels & Growth:** **Tech Lead** is often a senior position an engineer grows into after proving themselves as an excellent developer and system designer. Beyond Tech Lead, paths might include **Principal Engineer/Architect** (overseeing architecture across multiple teams or the whole company) or transitioning into **Engineering Management** (if also interested in people/project management). In an early-stage startup, the Tech Lead might become the **CTO** or a senior architect as the company scales, focusing on long-term technology strategy. Growth for a Tech Lead is demonstrated by their expanding impact: initially, they design for one team or product; later, they might influence architecture company-wide, mentor multiple teams, and evaluate new product technical directions. They’ll also be involved in strategic decisions like build vs buy, scaling strategies for the platform, etc. In terms of leveling, many companies use Staff Engineer, Principal Engineer beyond Tech Lead, which indicate larger scope and influence (like cross-team architecture, setting engineering-wide standards, etc.). A Tech Lead who continues coding and leading technically could reach those principal levels by consistently delivering robust architecture that enables the company’s growth.

***Signals of excellence****:* The product’s architecture is adaptable and has supported adding features quickly without major rewrites - that’s a huge sign the Tech Lead did it right. When there were surges in users or data, the system held up (or needed minimal tuning) because foresight in design accounted for it. Fellow developers say things like *“Our Tech Lead’s designs are always clear and make our job easier”* or *“We rarely have to say no to a feature due to technical limitations, because our architecture is solid.”* The Tech Lead also likely reduced the bus factor by spreading knowledge; multiple team members understand the system’s design, not just the Tech Lead. Code quality is consistently high - few critical bugs, a clean architecture where new devs can ramp up quickly because it’s well-structured. The Tech Lead is also often the one solving the gnarly problems that arise - their technical depth shows when an unexpected issue comes up in production and they can dive in, pinpoint the cause, and guide a fix quickly, earning respect from the team and demonstrating technical heroism when needed. But more sustainably, their excellence shows in the absence of crises: things *just work* and scale, which is the result of their behind-the-scenes work. Additionally, an excellent Tech Lead raises the bar for the whole engineering team’s skills - you’ll see team members adopting better practices, learning new techniques, and overall technical competence rising, driven by the Tech Lead’s influence. They also engage with the broader tech community (e.g. writing blog posts, attending architecture meetups, etc.) keeping the company at the cutting edge in relevant areas - which can help attract talent and recognition.

*Failure modes:* A Tech Lead can fail by **over-engineering** - designing an overly complex system that is a poor fit for the team’s size or the product’s current needs. This often shows up as lots of effort spent building generic frameworks or abstraction layers that aren’t actually needed yet, slowing down feature development dramatically. Another failure is **not documenting or communicating architecture** - if only the Tech Lead understands how things work and they become a bottleneck, that’s a single point of failure. Also, a Tech Lead who **ignores input** can cause friction; if they always impose their ideas and dismiss others, the team may follow grudgingly or workaround decisions, leading to inconsistency and morale issues. Conversely, a Tech Lead who **doesn’t assert enough** (e.g. lets a quick-and-dirty hack go in without pushing for a later fix) might see the architecture erode because they weren’t guarding it. There’s also **analysis paralysis** - constantly debating tech decisions and delaying implementation, perhaps wanting perfect information or consensus; startups need to ship, so a Tech Lead must balance deliberation with decisiveness. Performance issues or outages due to architectural oversight (e.g. they didn’t plan for thread safety and the app crashes under load) also indicate a Tech Lead slip - while not everything can be foreseen, repeated incidents that trace back to design flaws mean the Tech Lead might be in over their head or not paying attention to critical quality attributes. If the tech stack becomes outdated or the team misses out on tooling that could significantly improve productivity because the Tech Lead is resistant to change, that’s another fail (stagnation). Lastly, if developers are afraid to touch parts of the system because “it’s too complicated” or “only X understands it,” that’s a Tech Lead failure to simplify and disseminate knowledge. Essentially, if the architecture starts hindering rather than helping, the Tech Lead needs to course-correct.

**Weekly Action Plan for Tech Lead:**

* **Start of Sprint / Planning:** Review all upcoming user stories from a design perspective. If something needs an architectural decision, make sure to address it *before* implementation starts. You might write a quick design note or have a huddle on the first day of the sprint for any stories that need it (“Let’s talk through how to implement feature X - which module, any new classes or API contracts?”). Ensure that any enabling work (like provisioning a new server, or researching a library) is accounted for in the plan. If the PM/Dev Lead planned a lot of scope, confirm that it doesn’t require technical work beyond what’s visible; if it does, speak up early.
* **Design/Code Review Sessions:** Schedule time mid-week to do an in-depth review of either the design of an ongoing epic or the code of a critical component being written. For example, if a developer is implementing a new payment workflow, maybe by Wednesday you check in on their branch or design to ensure it’s aligning with security and architecture guidelines. Offer adjustments if needed while it’s still early enough. If multiple devs are working in parallel, hold a brief “architecture sync” where each dev explains their approach for their piece, and you confirm it all integrates well (this prevents last-minute integration surprises).
* **Technical Debt & Improvements:** Each week, try to tackle or at least plan one small tech improvement. For instance, spend a couple of hours updating a library, improving CI config, or writing a script to automate something. If it’s too busy, at least create a ticket for a tech debt item that you noticed and put it in the backlog. Keep the tech debt list visible. Perhaps allocate Friday afternoon as “10% time” for the team to work on cleanup or learning - lead by example by using that time yourself on an improvement task and encouraging others.
* **Mentor & Learn:** Have a one-on-one (or one-on-two) session with a developer to discuss a technical topic. It could be reviewing how a recently merged feature was implemented (discuss what was good, what could be improved) in a blameless way, purely as a learning exercise. Or pair program on something the developer is struggling with, like optimizing a query. Also devote a bit of time to your own learning: read an article or two on technology relevant to your product, or check documentation of a new version of a framework you use. Staying sharp is part of your job, and sharing relevant findings with the team is even better.
* **End of Week Reflection:** Look at any production metrics or alerts from the week. Did you see any performance warnings, error spikes, etc.? If yes, analyze them now (don’t wait for a crisis). For example, if memory usage is creeping up release over release, plan a fix (maybe a memory leak to plug). Check the logs - are there frequent errors or exceptions that need attention? This preventive maintenance is key. Communicate anything noteworthy to the team or put it on the backlog. Also, before the next week starts, glance at what’s next on the roadmap - if a big feature is coming in the next sprint or two, maybe you need to start preparing (e.g. reading up on an API, drafting a design doc to review with the team on Monday).
* **Ad-hoc Firefighting:** If at any point in the week an urgent issue comes up (like a prod bug or a major technical hurdle in development), jump in to help solve it. For production issues, lead the investigation (coordinate with DevOps if needed). For development hurdles, maybe a dev finds the architecture doesn’t support a requirement well - assemble relevant people and brainstorm a solution or workaround. Keep a cool head and guide the troubleshooting systematically. After resolution, note if there’s a root cause to fix long-term (e.g. design change, more tests) and take ownership of that as needed.
* **Communicate Upwards:** At least once a week, update the CTO or Eng Manager (if you have one above you) on the technical status - “We implemented X, found Y challenge but solved it like this, next week planning Z.” This keeps leadership confident in the tech side and gives opportunity for them to provide input or resources. In a small startup, this might just be a chat with a founder. Also discuss any technical debt or risks that might need more time or hiring to address in the future, keeping them in the loop well before it’s critical.

### Software Developer (Dev)

**Mission:** *Build and deliver software features with high quality, collaborating with the team to turn requirements into reality.* Developers (engineers, programmers) are the ones writing the code day in and day out that makes the product work. In an early-stage startup, each developer can have significant impact on product outcomes. Their mission is to produce working software that meets specifications, is reliable, and is maintainable, while continuously improving their craft.

**Key Responsibilities:**

* **Implement Features:** Write code to implement user stories and requirements assigned in each sprint. This involves understanding the acceptance criteria, asking clarifying questions if needed (don’t assume - communicate with the BA/PM/Tech Lead to resolve ambiguities), and then designing, coding, and unit-testing the solution. A developer should follow the team’s coding standards and best practices while doing so. That includes structuring code properly, writing clear and concise logic, and handling edge cases as known. They should also integrate their code with the existing system - ensuring not to break existing features. If the feature is user-facing, possibly pair with designers or product folks to get it right. Essentially, deliver the story to **Definition of Done**, which typically includes coding, code review, and initial testing.
* **Bug Fixing and Maintenance:** Investigate and fix software defects, whether found by QA, reported by users, or discovered through monitoring. This requires debugging skills - reading logs, replicating issues, tracing through the code to identify root causes. A developer should not just patch symptoms but aim to resolve the underlying problem. They also take care to verify the fix (write a regression unit test or replicate the scenario to ensure it’s fixed). Maintenance also covers tasks like updating dependencies, improving small parts of code they touch if they see an obvious improvement (the “Boy Scout rule” of leaving code cleaner than you found it), and contributing to reducing technical debt in the areas they work on. In a startup, developers have to be resourceful in debugging because there may not be fully established processes - using any tools available (debuggers, print logs, etc.) to track down issues quickly.
* **Code Reviews & Collaboration:** Participate actively in code reviews of peers’ work. This means reviewing pull requests in a timely manner, providing constructive feedback on logic, style, test coverage, and alignment with requirements. By reviewing others’ code, developers maintain code quality and also learn from each other. Additionally, collaborate during design discussions - even if a Tech Lead or senior dev is leading the design, developers should engage, ask questions, and voice concerns or ideas. Within the team, help each other: pair programming on complex tasks, sharing knowledge of areas of the code (each dev often has deep knowledge in some parts; helping others understand those is valuable). In daily stand-ups, communicate what you’re working on and where you might need help or might be blocked. Being a team player is critical - especially in startups where there’s no room for siloed work or “my part is done, I don’t care about the rest”. If needed, a developer might pick up tasks slightly outside their comfort zone (e.g. a frontend dev helping on a simple backend fix) to move the team forward, learning as they go.
* **Testing and Quality Assurance:** Developers are responsible for initial quality of their code. This means writing unit tests and possibly integration tests for their features to ensure the logic is correct. They also perform self-testing - running the application to verify the feature behaves as expected, covering main flows and basic edge cases before handing over to QA. If an issue is found, they fix it upfront. They should adhere to any DoD item like “unit tests written” or “100% acceptance criteria passed” before marking a story as done. In some teams, developers may also write or update automated UI tests or API tests, or at least collaborate with QA on them. And when QA finds bugs, a good developer treats it professionally - not as a personal affront - and fixes the bug promptly, thanking the QA for catching it. Quality is a shared responsibility: developers ensure they don’t throw code “over the wall” to QA that obviously doesn’t work or hasn’t been checked. Also, if production issues occur, developers participate in troubleshooting and apply fixes or mitigations. They learn from each bug - possibly adding additional tests or checks so that class of bug doesn’t recur.
* **Continuous Learning and Improvement:** Because startups evolve quickly, developers need to continuously improve their knowledge of both the codebase and the technologies in use. They should be actively learning: reading documentation, trying out new tools on the side, following industry best practices. When they discover a more efficient method or a best practice (for example, a new library that could reduce lots of custom code, or a refactoring pattern that simplifies a module), they bring it to the team’s attention. They also reflect on their own work patterns - e.g. if they notice that lack of upfront planning caused them to rewrite code, next time they might sketch a design first or seek a quick design review from a senior colleague. In terms of improvement, developers should respond to feedback (from code reviews, peer comments, etc.) constructively and incorporate that feedback in future work. The expectation is each developer’s output quality increases over time as they gain experience with the system and learn from mistakes. In a broader sense, developers can contribute to improving team processes as well - maybe they suggest a tweak in how deployments are done if they see an inefficiency, or volunteer to write a script to automate a manual step, etc. Proactiveness is highly valued in a startup environment. Developers shouldn’t just wait for someone to tell them exactly what to do - they take ownership of their tasks and also think about what else can be done to make the product and code better.

**Scope & Boundaries:** A developer’s scope is to implement and deliver individual tasks and contribute to overall product increments. They have ownership of the code they write, but *collective* ownership of the codebase - meaning they should feel responsible for the whole product quality, not just “my feature”. That said, boundaries are important: a developer should follow the guidance of Tech Leads/Dev Leads for consistency - e.g. use the approved tech stack, follow established architecture patterns, and not introduce random technologies without discussion. If they strongly feel a different approach is better, they can and should propose it, but they shouldn’t go rogue and implement it without buy-in. Developers typically do not decide *what* features to build (the product team does) or *when* deadlines are (PM/leadership sets that), but they can estimate and voice if something seems infeasible in a timeline, providing input that might adjust those decisions. They shouldn’t gold-plate or significantly alter feature scope on their own; any changes to requirements should loop back to the BA/PM for confirmation. Another boundary: **don’t neglect tests or documentation** just to go faster - in many teams, writing code means also writing associated tests/docs; leaving those out isn’t considered done. Also, while developers should help each other, one developer is not solely responsible for another’s work - avoid stepping on toes by rewriting someone else’s code entirely without consultation (code review is the forum for feedback). And importantly, developers must respect the production environment boundaries - e.g. no one deploys changes or hotfixes to production without following protocol (like code review, CI/CD pipeline), because cowboy coding can risk the live system. In a small startup, protocols might be light, but the principle is communicate and coordinate when making changes that affect others or the live product.

**Key Collaboration Interfaces:**  
- **With Other Developers:** Peers on the team are your daily collaborators. You probably do pair programming or at least frequent Slack/Teams communication to discuss how to implement something. If you’re stuck, you ask a teammate for advice or search together. Sharing knowledge among devs is crucial - e.g., if you found a clever solution to a tricky bug, you might send it in chat *“FYI, to fix the timezone bug I had to use library X - keep that in mind for date handling.”* Devs also often review each other’s code. It’s a give-and-take: review others as you’d like yours to be reviewed - respectfully and helpfully. In an early-stage startup, the devs may also share non-coding responsibilities like setting up dev tools, maintaining build scripts - they work together to keep the dev process smooth. There’s usually little specialization; everyone jumps where needed. This fosters a camaraderie - you succeed or fail as a team.  
- **Development Lead / Tech Lead:** Expect daily or near-daily interactions. The Dev Lead might be assigning tasks or checking in, and you update them on progress or issues. Use them as a resource - ask when in doubt about priorities or when you need a decision (like “Should I refactor this now or leave it?”). They also do your code reviews; take their feedback seriously and discuss if you disagree. A Tech Lead might give you a design to implement - make sure you understand it; if something seems off or you have a suggestion, don’t be silent - voice it. Essentially, communicate well with leads: let them know if you’re ahead or behind, if you encounter scope changes, etc. They rely on devs to surface issues early. Also, when they propose best practices, try to adopt them; if you find it challenging, ask for guidance. It’s a two-way street: leads mentor you, but you as a developer provide the raw implementation and ground reality feedback to them. This collaboration ensures that architecture and execution stay aligned.  
- **Business Analyst / Product Manager:** While not as constant as with technical colleagues, devs do interact with BAs/PMs whenever requirements need clarification. If a user story’s acceptance criteria are unclear, a developer should reach out to the BA: *“Hey, in scenario X what should happen? The story doesn’t say.”* It’s far better to ask than assume wrongly. In some setups, devs attend backlog grooming or sprint planning with BAs/PMs - that’s a chance to ask questions and also give input on if something seems technically risky or related to an existing component (e.g. “this request is similar to something we built last month, should we reuse that?”). Devs might also give demos of implemented features in sprint reviews to the PM and stakeholders, explaining how it works and confirming it meets the need. That’s a collaborative moment where the PM might give additional feedback (“Can we tweak this?”). Also, if a developer spots a possible improvement or a bug that’s user-facing beyond the immediate task, they should inform the BA/PM to decide if it’s critical or can be backlogged. In early startups, developers often wear a bit of product thinking hat too - they sometimes suggest small UX improvements or identify misalignments (“this design is hard to implement but if we change it slightly it’ll be much easier - is that acceptable?”). Having that dialogue with product folks is valuable to find pragmatic solutions.  
- **Quality Assurance:** If there are dedicated QA engineers, developers work closely with them. The QA might create test cases; the developer can preempt some by running through those scenarios themselves. When QA finds issues, developers and QA should communicate clearly: how to reproduce, what was expected vs actual. Avoid adversarial attitudes; both share the goal of quality. In triaging bugs, a developer helps assess severity and priority with QA input. Often a developer will fix a bug and then talk to the QA who found it to ensure it’s truly resolved, maybe asking them to re-test the specific scenario. In some teams, devs and QAs do joint sessions like a *“buddy testing”* where a dev walks a QA through new functionality to point out areas of risk or special cases, and the QA gives immediate feedback. This can catch issues early. A developer should treat QA as a collaborator who helps ensure their work is solid, not as someone just to “find my mistakes.” If no separate QA (common in very early startups), then developers might peer test each other’s features, essentially fulfilling the QA role as a team - which again emphasizes working together for quality.  
- **DevOps / Operations:** If the startup has someone focusing on infrastructure or if developers themselves rotate in ops duties, collaboration here ensures smooth deployments. A dev should understand the deployment process (even if they don’t run it) and ensure their code and config works in those environments. If a dev’s feature requires, say, a new environment variable or file migration, they must communicate with the person handling deployments. Developers might also help debug environment-specific issues - e.g. something works on localhost but not on staging, they’ll work with ops to figure out server differences. If on-call rotations exist (devs might share pager duty for incidents), collaboration means handing off knowledge - e.g. if you deploy a risky change on Friday and someone else is on-call over the weekend, give them a heads up on what to watch. In early startups, devs are often devops too - setting up CI pipelines, writing Dockerfiles, etc., in which case they collaborate as a team to maintain those. The main point is: developers should not throw code over to ops with “it works on my machine” - they have to engage in making it work reliably in all environments. This might involve writing build scripts, containerization, etc., collectively deciding those with more ops-minded colleagues.  
- **Customers/Support (Indirectly):** Typically devs aren’t front-line with customers, but they might work with customer support or use customer feedback. For example, if a user reports a bug, a support agent or PM might relay it to a developer to fix. The developer may ask for additional info (logs, user data) and support facilitates that. Sometimes devs do get on a call for technical discovery with a client (especially B2B or if the startup is small) - e.g. to assist integration. In such cases, devs must communicate clearly and non-verbosely with non-tech or semi-tech folks, often translating issues or needs. This is less frequent but can happen. Internally, if the startup is small, devs might also use the product and give feedback as quasi-users, or at least be aware of user pain points through team meetings. Being user-oriented helps - a developer thinking “How will the user use this? What could go wrong for them?” writes better code. So indirectly collaborating with the idea of the user through proxies (PM, support) is important.

**KPIs for Developers:** *(Often team-wide metrics apply, but here are ones a developer influences directly.)*

* **Task Completion Rate:** *Definition:* The number of tasks or user stories a developer completes (to done) per sprint or month, versus those assigned or started. This can be rough because complexity varies, but it gives a sense of throughput. *Target:* Consistently complete the tasks taken on, with maybe occasional spillover if things were underestimated. Essentially, a reliable personal velocity. *Anti-gaming:* Quality must be maintained; finishing many tasks but each with bugs is not true completion. Also, tasks sizes differ; it’s more a monitor of consistency. The intent is to encourage developers to accurately estimate and finish what they commit to, improving over time. A developer can use it to self-manage: if frequently not completing tasks, figure out why (underestimation? distractions? need help?). Note: this KPI is usually not public or comparative, more for personal improvement, because comparing devs by story count is not advisable (could promote cutting corners or avoiding hard tasks).
* **Code Quality & Review Feedback:** *Definition:* Could be measured by number of review comment iterations on a PR, or number of major issues found in code review for that developer’s contributions over time (with the idea that fewer issues over time = improving code quality). Also static analysis results specifically for their code (like lint warnings introduced). *Target:* Over time, the developer’s code requires fewer significant reworks from review, indicating they’re writing cleaner code initially. *Anti-gaming:* Ensure code reviews remain thorough. The developer should internalize common feedback so it’s fixed before review. This metric should be used as positive reinforcement (less “nitpicks” needed, good job) and guidance (if a lot of issues keep arising on their code, they may need mentoring). It’s tricky to quantify perfectly, but qualitatively leads notice if a dev’s code sails through reviews or always needs a lot of changes.
* **Defects Introduced:** *Definition:* Number of bugs in production or QA that are traced back to code the developer wrote. (This is usually team-shared, but can be looked at individually for coaching.) *Target:* Aim to minimize, e.g. trending toward zero critical bugs from one’s code. Everyone will introduce some bugs, but a good developer catches most before code leaves their desk. *Anti-gaming:* Encourage thorough testing by dev. Don’t punish for every bug (that creates fear), but use it to learn. If a particular developer’s changes frequently cause issues, they might need to slow down and test more or get design help. The developer should focus on improving their personal QA: writing unit tests, doing careful review. This KPI ties into the DORA “change failure rate” at a micro level; good engineering practices reduce failures from changes.
* **Commit Frequency / Size:** *Definition:* How often and how much the developer commits code. For example, average commits per day or average lines of code per commit. Ideal is frequent, small commits. *Target:* At least one commit most days (if actively coding) with logical groupings of changes rather than giant dumps or ultra-tiny commits. *Anti-gaming:* It’s not about lines of code (more is not better), but about incremental development. Many factors influence commit patterns, but a consistent flow shows the developer is integrating work steadily, which helps CI and feedback. If someone commits only one big batch at sprint end, that’s a risk. Encouraging smaller, frequent commits leads to easier reviews and merges. Developers can self-track - if they realize “I haven’t committed in 3 days because I was sitting on a big change,” they may consider slicing it.
* **Skill Development & Initiative:** *Definition:* A qualitative KPI - e.g. contributions to code outside of immediate assignments, learning new skills, taking initiative to improve something not explicitly assigned. For instance, did the developer in the last quarter pick up a new skill (front-end dev tries some back-end, etc.), or did they proactively fix a minor bug without being told, or improve documentation? *Target:* At least one notable self-driven improvement or upskilling effort in a quarter. *Anti-gaming:* This is about attitude and growth - it encourages developers to not just do the bare minimum. An example success: a developer notices page load is slow, profiles it, finds a minor fix, implements it - small but impactful initiative beyond their tasks. Or they enroll in an online course on a technology the team will need. It’s hard to measure, but leads/managers often recognize these behaviors. It’s part of being a strong startup developer - wearing multiple hats when needed and continuously growing. This “KPI” can be captured in performance reviews or weekly shout-outs more than a number. But it’s included to emphasize the expectation that developers in a startup context should proactively improve both the code and themselves, not just check tasks off.

**Career Levels & Growth:** A **Software Developer** in a startup might start as a **Junior Developer** (new grad or entry-level) then grow into **Mid-level Developer** (independent on most tasks), then **Senior Developer** (handles complex tasks, mentors others). From there, they might choose a track: becoming a **Tech Lead** or **Architect** (staying hands-on technical and leading design) or a **Development Lead/Engineering Manager** (leading people and projects). Some may specialize (e.g. becoming a Mobile Lead, or a Data Engineer if that need arises). In early-stage companies, developers have the opportunity to quickly take on more responsibility as the team grows - a capable mid-level dev today could be leading a small team in a year if the company scales and they show leadership. Growth is seen in: the scope of problems they can tackle (from small bugfixes to designing whole components), the level of guidance needed (from needing detailed direction to working independently and making decisions), and impact (from implementing well-defined tasks to proactively defining improvements and significantly accelerating the team’s output). Career signals also include mentorship - a senior dev helps juniors - and influence - e.g. a mid-level who becomes go-to person for a certain area or significantly improves team practices is showing senior traits. In startups, titles might be fluid, but the principle is the more value and reliability a developer demonstrates, the more leadership (technical or organizational) they can assume.

*Signals of excellence:* A great developer consistently delivers *working* code that meets requirements, with few bugs, and does so within estimated time (or communicates early if not). They have a solid understanding of the entire codebase, not just their piece, and contribute beyond their assignments - like helping fix urgent issues even if it’s someone else’s area, or optimizating code when they see an opportunity. They adapt quickly to new challenges - if tomorrow the product needs a blockchain integration (just as an example), they eagerly learn what’s needed and contribute. They receive code review comments well and you can see their code quality improving over months. Perhaps they even start giving insightful comments on others’ code, showing understanding beyond their own tasks. Other team members trust that if this developer is handling a task, it will likely be done right. In stand-ups, they raise important questions or ideas that improve understanding for everyone. When crunch time comes, they step up and help get things across the line, but also advocate for fixes or improvements after. Also, an excellent dev often has a mindset of ownership - they feel responsible for the product’s success, not just their code. So they might monitor after a release to ensure their feature works in prod, or suggest enhancements when using the product. All in all, excellence is marked by reliability, quality, proactivity, and teamwork.

*Failure modes:* A struggling developer might frequently produce code with bugs or that doesn’t quite meet the spec (missing acceptance criteria). They might require a lot of hand-holding or rework. If deadlines are often missed due to their tasks or their tasks get re-assigned to others to complete, that’s an issue. Another sign is poor collaboration: maybe they don’t communicate when stuck, resulting in big delays or subpar implementations because they didn’t ask for help or clarification. If they are defensive in code reviews and never incorporate feedback, their growth stagnates and team cohesion suffers. Also problematic is **over-engineering or perfectionism** to the point of not delivering (the dev who refactors endlessly or chases an ideal solution and misses timeframes). Or the opposite, **quick and dirty coding** to close a task on time but leaving a wake of issues for QA or others to fix - this erodes trust. Not taking responsibility is another failure mode: e.g. “It works on my machine, not my problem” or blaming requirements every time instead of improving understanding. A developer who doesn’t improve or even regresses in skill/attitude could become a bottleneck. Perhaps they avoid learning new parts of the system and only stick to what they know, forcing others to pick up the slack on new tech. Also, if a developer has poor testing discipline, causing lots of production bugs repeatedly, it’s a serious problem in a small company where quality issues can be life-or-death. Essentially, if a dev’s presence isn’t net positive on output or if they actively harm team dynamics (poor communication, ignoring standards), those are failure states. Such cases need coaching or, if not fixable, a role change or exit, because early-stage startups can’t carry someone who consistently underperforms - the impact is too large on a small team.

**Weekly Action Plan for Developers:**

* **Beginning of Sprint:** Make sure you fully understand the tasks/stories you’ve been assigned. If the sprint planning meeting didn’t answer some questions, seek out the BA or Tech Lead immediately to clarify. Perhaps write down a quick approach for each of your tasks and run it by the Tech Lead or a senior dev if you’re unsure (this can prevent going down the wrong path). Set up your development environment or feature flags for the new work - e.g. if working on a new module, scaffold any needed project structure early. Basically, get organized so you can hit the ground coding.
* **Daily Routine:** Aim to write and commit code in small chunks each day. For example, before lunch, implement a sub-function or model and commit it with a clear message. After lunch, maybe integrate that with the API and commit again. Don’t wait days for a giant commit. Each day, also run the unit tests (or relevant subset) to ensure you haven’t broken anything - keep the build green. If you find yourself stuck on a bug or decision for more than say 1-2 hours, call over a teammate or post in the team channel describing the issue - a fresh set of eyes can save you time. Keep communication open: if you realize something might take longer than expected, inform the Dev Lead early (don’t surprise them a day before deadline). And reciprocally, if you finish something early, either start writing extra tests, do a self-QA, or pick up another team task - always ask how you can help.
* **Mid-week Checkpoint:** Review your progress mid-sprint. Are you about halfway through your tasks by roughly halfway of sprint (adjust if tasks are not equal size)? If behind, prioritize and possibly simplify some implementation to catch up, but inform leads if any de-scoping happens. It’s also a good time to do some cross-testing: maybe check a colleague’s new feature in the test environment while they check yours - you might catch issues early (buddy testing). Additionally, attend any mid-sprint backlog grooming or design sessions to prep for upcoming work. It’s easy to zone out if it’s about future tasks not yours, but pay attention - you might be the one coding it soon, and you can contribute technical insights now.
* **Before Code Reviews:** When you believe your code for a story is done, do a personal review first. Run all tests, perhaps do an exploratory test of the feature end-to-end in your local environment. Use a linter or formatter if not automated, to keep style consistent. Try to read through your diff once - you’d be surprised how many small mistakes or improvements you can catch (like leftover debug prints, or a poorly named variable) before others see them. This makes the actual code review smoother. Then create the pull request with a clear description of what and why. Be available to respond to review comments promptly - don’t toss it over the wall and disappear. If reviewers suggest changes, get those done in a timely manner (same day or next day). Learn from each comment for next time. If there’s discussion needed, have it in the review or a quick call. The sooner the PR is approved and merged, the better to avoid integration issues, so treat it as high priority. Likewise, allocate some time each day to review others’ code - it’s part of the job.
* **Testing & Handover to QA:** Once your code is merged to a test/staging environment, perhaps run through the main user flows yourself there. Make sure any config or data needed is set (e.g. “Did I tell QA they need to have feature flag X on to see my feature?”). Work with QA: maybe demo the feature to them or write a brief note on what to focus on. This isn’t to bias testing but to ensure they know the scope. If any known limitations exist, point them out (and have them documented). When QA reports bugs, don’t be defensive - prioritize fixing them. Try to fix minor bugs immediately (same day), so QA can re-test without losing momentum. For more complex bugs, talk through with QA to understand exactly what’s happening, then solve it methodically. Aim that by the end of the sprint, your feature is truly done-done (coded, reviewed, tested, fixed).
* **End of Sprint:** During the sprint review, if you’re asked to demo what you built, prepare a straightforward walkthrough. Highlight the acceptance criteria and show they’re met. Also mention any edge cases or improvements that were made. Accept feedback from stakeholders or PM; if minor adjustments are requested and they’re quick, perhaps do them immediately or at least size them for next sprint. In the retrospective, reflect on your work: did anything hinder you? Perhaps lack of test data, or unclear story, or a development tool issue. Bring it up constructively: *“It took me a while to test the payment feature because I didn’t have test credit card data; maybe next time we set up better test data upfront.”* Offer suggestions if you have them. This helps the team improve and shows you think beyond just your coding. Also, retros are a time to appreciate - maybe mention *“I appreciated Alice’s help debugging that race condition”*, reinforcing team collaboration culture.
* **Continuous Learning:** Outside the direct task work, set aside a little time each week for learning something new or improving a skill. It could be reading an article on a library you use, experimenting with a small side project that teaches you a concept, or even contributing a tiny fix to an open-source project your startup relies on (which also counts as giving back and learning). If your company has access to courses or books, use them. You can even propose a short “lunch and learn” to share knowledge with teammates if you discovered something useful. This not only grows your expertise but also signals initiative. While it can be hard in a busy startup to find time, even an hour a week is good. Over months, this adds up and you’ll notice yourself applying new techniques to work tasks, becoming faster or writing better code. It’s an investment in yourself that also benefits the company in the long run.

### Quality Assurance (QA) Engineer

**Mission:** *Ensure the product is delivered with the highest possible quality by finding and preventing defects, verifying that all requirements are met, and guiding the team’s testing efforts.* The QA’s goal is to act as the last line of defense (and ideally, an embedded partner throughout development) to catch issues before customers do, and to uphold the quality standards of the software. In a startup, QA engineers help introduce process and thoroughness in testing without losing speed.

**Key Responsibilities:**

* **Test Planning & Case Design:** For each feature or user story, create a **test plan** that covers how the functionality will be verified. This includes identifying test scenarios (positive flows, negative inputs, edge cases, error conditions) that need to be checked. The QA writes detailed **test cases** or checklists covering these scenarios - often in a test management tool or even a spreadsheet if that’s what the team uses. For example, if the feature is a login form, test cases would include correct credentials, wrong password, SQL injection attempt, etc. They also think about *non-functional* aspects if relevant (performance, security, usability). By having a solid set of test cases ready as development starts finishing pieces of the feature, QA ensures systematic coverage. In startups, a QA might need to write tests even when requirements are changing - so they maintain flexibility, updating test cases as stories evolve. If the team follows BDD (Behavior-Driven Development) or given/when/then acceptance criteria, the QA can help flesh those out. Essentially, QA makes sure that for every requirement, there’s at least one corresponding test case that will confirm it.
* **Manual Testing & Exploratory Testing:** Execute test cases on the application, especially new features in the staging/QA environment. QA engineers systematically go through the planned test cases (manual testing) marking pass/fail, and logging any defects found in the process. They pay attention to details - not just whether the main function works, but also UI alignment, input validation messages, etc., to ensure a polish level. Beyond scripted cases, QA also performs **exploratory testing** - creatively using the application in unplanned ways to try to uncover unexpected issues (for instance, rapid clicking, trying unusual data sequences, partial submissions, etc.). Good QA testers think like a user and also like a “hacker” to find issues normal flows might not reveal. When they find bugs or any deviation from expected behavior, they promptly file clear bug reports - with steps to reproduce, what happened vs what expected, screenshots or logs if possible. They prioritize the bugs (critical vs minor) and communicate any showstoppers immediately to the team (so devs can perhaps fix before a release). In iterative startups, QA might do *smoke testing* of each new build and *regression testing* of core features before each release to make sure nothing that worked before got broken by new changes.
* **Test Automation (if applicable):** Many QA engineers also create automated tests to speed up regression checks. This could involve writing **automated UI tests** (using tools like Selenium, Cypress, etc.), **API tests** (with tools or custom scripts), or **performance tests** (using JMeter, etc.), depending on skills and need. In an early startup, there might not be a full test automation framework yet - a QA can set the foundation by identifying which tests to automate first (usually the critical path that is tedious to redo manually each time). They might write scripts that, say, run through login and key screens and assert no crashes. If the company already has some automated tests in the CI pipeline, the QA maintains them - updating tests when UI or flows change, adding new tests for new features, and investigating test failures (to distinguish flaky tests vs real product issues). Automation is a force multiplier: it catches regressions quickly and frees QA time from repetitive checks so they can focus on new functionalities and exploratory testing. Even if the QA isn’t a coding expert, they often contribute by writing test scripts or working with developers to integrate test tools in the build process.
* **Defect Management & Verification:** Manage the life cycle of bugs from discovery to resolution. That means after logging a bug, tracking its status - ensuring it gets assigned, fixed, and retested. When developers claim a bug is fixed, the QA re-tests the exact scenario to confirm the fix (this is **verification**). If it passes, they mark the bug as closed; if not, they reopen it with feedback that the issue persists or changed. QA often also does **regression testing** around the fix area - e.g., if a bug in the payment module is fixed, test other payment flows to make sure the fix didn’t break something else. They maintain the bug list, sometimes adding notes or grouping duplicates, and highlighting any high-severity issues that must be fixed before release (they serve as quality gatekeepers; if there’s a critical bug, they should advocate not to release until it’s addressed or a workaround is in place). Additionally, QAs can analyze the pattern of defects to provide insights: e.g. “We’re seeing a lot of issues in the sign-up flow; maybe we need to refactor that or have more unit tests there.” They might generate simple reports on bug counts, severity, etc., to keep the team informed of quality trends.
* **Quality Process Advocacy:** Establish and refine the testing and release processes to improve quality culture. This might include introducing a **Definition of Done** that includes QA sign-off (ensuring the team formally acknowledges QA’s role in completeness), setting up **QA environments** (if one doesn’t exist, push for a stable testing environment that mimics prod), and teaching developers some testing practices (like encouraging devs to do basic acceptance testing before marking a story ready for QA). They might run **test case reviews** with developers to see if any scenarios are missing from the test plan - which helps share testing mindset with devs. QA can also drive use of tools like bug tracking systems, and maintain **test data** (ensuring there’s a set of test accounts or seed data for consistent testing). In early startups, formal processes might be minimal, so a QA brings needed discipline: e.g., they might propose that before a major feature release, the team does a brief *bug bash* where everyone including devs tries to break the feature - and QA coordinates that. They may also keep **quality metrics** visible (like number of open bugs by priority) to raise awareness. Overall, QA serves as the voice of quality in planning and retrospectives - reminding the team about testing needs, upcoming risks (like “we have a lot of changes next sprint; we should schedule extra regression time”), and continuously refining how the team approaches quality assurance. They strive to shift testing left - meaning involve QA considerations earlier in development, so fewer bugs reach the end.

**Scope & Boundaries:** The QA’s scope is **quality of the product** - which spans functional correctness, usability to some extent, and reliability of new deliveries. They do not make product decisions (that’s PM/BA) but they *can* question requirements if something seems inconsistent or untestable (“This user story doesn’t specify validation limits - can we clarify?”). Their boundary is that they shouldn’t dictate *how* developers implement things (not their role to design code) but they can certainly highlight if something is consistently leading to bugs. QAs are not “gatekeepers” in an adversarial sense; they are collaborators. A boundary to manage is time vs thoroughness: in a startup, testing everything exhaustively might not be possible before a release; the QA must prioritize tests based on risk and communicate what was not fully tested. It’s understood that not every minor issue can be fixed; QA should focus on critical quality issues and not insist on perfection to the point of blocking progress - they must gauge when to be firm (e.g. a security issue must be fixed) vs when to note a minor UI quirk and let it slide to be fixed later. They should respect development constraints and work with the team to find solutions (like accepting a small risk if timeline demands it but planning to address it soon). Another boundary: QAs don’t typically write production code (though they might write test code); they shouldn’t directly change code unless it’s minor and agreed (developers fix bugs, QA verifies, rather than QA themselves altering product code which could lead to confusion or new issues). However, in some agile teams, QAs and devs pair on fixing a bug - which can be fine as long as roles are clear. QAs should maintain objectivity - they advocate for the user’s perspective, and sometimes that means diplomatically pushing back on releasing if quality is not there. But final decisions may be made by product or management; QA provides input and risks (“We have 3 critical bugs unresolved - releasing now is risky”), and ideally management heeds it, but QAs might not have formal veto power. Regardless, their professional responsibility is to clearly state quality concerns. One more boundary: ensure test environments and production are handled carefully - QAs might have access to prod logs or databases to verify fixes, etc., but they should be cautious not to disturb real data or privacy. They should coordinate with devops if something needs to be tested in prod (like a hotfix verification) and follow any protocols.

**Key Collaboration Interfaces:**  
- **Developers:** This is the closest collaboration. QAs and devs should work as a tandem rather than adversaries. For each story, a QA may talk with the dev about how the feature is implemented to design better tests (“Can I see how you structured the input validation? Okay, I’ll test those boundary cases.”). When bugs are found, QA provides clear info to devs so they can fix it - sometimes they might even reproduce the issue together so the dev can debug real-time. A great practice is **QA involves devs in testing** - e.g. pair up to do exploratory testing: the dev might see something QA misses and vice versa. Also, devs involve QA by having them check things early; for instance, a developer might deploy a feature to a test environment and ask QA for a quick sanity check *before* finalizing - QA can catch obvious issues earlier (cheaper to fix). The QA should feel comfortable to ask a dev “Can you explain this logic? I want to ensure I test it fully.” and devs should be open in explaining. Communication is critical: QA announces new bugs in a way the dev can quickly understand severity; devs update QA when a bug is fixed or if a requirement changed. **Mutual respect** is key - both roles want a quality product. In retrospectives, if QAs feel rushed (like “didn’t have time to test X”), devs should hear that and maybe adjust estimation or process to give QA more time. QA also learns from devs - e.g. understanding system internals helps testing - and devs can learn from QA how to think of edge cases, which they then incorporate into their unit tests or development process. A collaborative environment means sometimes devs help test and QAs might assist in troubleshooting, all working toward the same goal.  
- **Product Manager / BA:** QAs work with BAs/PMs primarily around clarifying acceptance criteria and ensuring the *requirements are testable*. If a requirement is unclear or contradictory, QA will loop back to BA for clarification, possibly pointing out scenarios not considered (“What should happen if user input exceeds 100 characters? The requirement doesn’t say.”). They also confirm the *acceptance criteria* in stories are sufficient and, if not, may propose additional ones. In planning or grooming meetings, a QA might ask the BA/PM questions from a quality perspective, which can sometimes even uncover requirement gaps (e.g. *“We say the app supports multiple users - do we have a requirement on what happens when two users edit at same time? If not, we might have an undefined behavior there.”*). QAs also communicate to PMs the quality status prior to release - if there are known issues, QA makes sure PM is aware (no one likes surprises from customers). This helps PM make go/no-go decisions. PMs often rely on QA to validate that “done” is truly done - a PM might ask QA “Are we confident this feature is ready for customers?” and QA should give an honest assessment. For instance, QA can say *“We tested the main flows thoroughly, but not all edge cases due to time; overall it’s stable with one minor issue left.”* That info helps PM decide to release or wait. Essentially, QA gives PM a risk vs quality snapshot.  
- **Tech Lead / Dev Lead:** QAs coordinate with tech leadership for things like environment readiness, test automation integration in CI, and to highlight any recurring quality problems that might require tech lead attention (like a module with frequent bugs - maybe needs refactoring). The Tech Lead might ask QA for input on quality metrics or to prioritize testing on certain risky features. QA might say *“The new architecture for search was complex; we should allocate extra testing time here”* to which the Tech Lead can respond by scheduling accordingly. Dev Leads often enforce that no story is complete without QA sign-off; QAs ensure to sign off in a timely manner and voice if they feel pressured to sign off without enough testing (the Lead should mediate that). If the team lacks full automated tests, a Tech Lead might work with QA to decide on a framework - QA will then collaborate to implement it. Additionally, if performance or security testing falls under QA and requires developer support, tech leads plan that. For example, QA says *“We want to do a load test but need a production-like environment”*, Tech Lead/DevOps arranges that. So, QA and leads engage to improve the overall QA process, tooling and resourcing. Another scenario: In a root cause analysis (post-mortem) of a production issue, QA participates with the Tech Lead to understand why it wasn’t caught and how to improve processes (maybe adding a test case or updating the monitoring). It's a partnership aimed at continuous improvement of quality practices.  
- **Designers/UI/UX:** If the startup has a UX/UI designer, QA should include UI/UX aspects in their testing. That means QA checks alignment with design mockups, correct fonts/colors, responsive behavior, etc. The QA may collaborate by verifying with the designer on whether a certain behavior is as intended when uncertain. For instance, *“The tooltip text is cut off in the UI - is that the design or a bug?”* If it’s a bug, QA logs it; if design changed but wasn’t updated in requirements, QA helps catch that discrepancy. Also, for usability issues - QA might notice something like *“After completing the form, there’s no success message, which might confuse users”* - they can bring that up, either directly to the designer or via the BA/PM, to see if it’s intended. QAs often act as a semi-user advocate (though not a substitute for UX research) and can highlight obvious usability problems. Collaborating with designers ensures that the product not only works right but also looks and feels right. If style guides exist, QA can refer to them. Sometimes QA might even have to test on multiple browsers or devices - for that they might align with designers on which platforms are priority (e.g. must work on Chrome and Safari, or on iOS and Android for a mobile app - designers usually define supported platforms with PM). QA ensures those platforms are tested. And if issues are found, sometimes designers provide the expected behavior (like how it should have looked), which QA includes in bug reports. In summary, QA helps ensure that the implementation and the design are consistent and high-quality, collaborating by asking for design references and confirming uncertainties.  
- **DevOps/Support:** QAs sometimes interface with DevOps (if separate) for test environment stability, test data resets, etc. If a QA environment is down or needs a deployment, QA coordinates with DevOps to get new builds deployed for testing. QAs might also be the ones verifying a hotfix in production - working with DevOps to deploy to a staging environment first, test it, then giving the green light to deploy to prod. In smaller startups without dedicated DevOps, QA might rely on developers to deploy test builds. As for Support (customer support teams), QAs can be a bridge: if support tickets come in describing bugs, support forwards them to QA to reproduce and confirm before devs jump in. QA reproduces the issue in test or prod, documents it, and logs a bug with all necessary info (taking the burden off support and giving devs a clear report). QAs might even verify support’s claims like “customer says data is missing” by checking logs or database (read-only) to see what happened, then involve devs if it’s a true bug. This collaboration ensures customer-reported issues are handled efficiently - support gets confirmation and eventually an answer to give back once dev fixes it. QA keeps track of these externally reported issues too and might prioritize testing around those areas more in the future.  
- **Regulatory/Testers (if any external):** If the domain is something like healthcare or finance where external audits or certifications are needed, QA might work with compliance folks or external testers during UAT (User Acceptance Testing) phases. They ensure all test evidence is captured properly for auditors (like saving test result logs) and that test cases map to regulatory requirements. This is less common but possible in startups tackling regulated industries. Then QA’s role extends to handling documentation and communication with those external quality assessors.

**KPIs for QA:** *(Focus on quality outcomes and process efficiency.)*

* **Bug Detection Effectiveness:** *Definition:* Number of bugs caught in testing (pre-release) vs bugs reported by end users after release. This can be a ratio or percentage: e.g., if 50 bugs found internally and 5 escaped to users, that’s 91% caught internally. *Target:* High internal catch rate, e.g. > 90% of total bugs are caught before release (meaning minimal escape rate). *Anti-gaming:* This depends on users reporting issues too (some might not be reported). But as a trend, if post-release bugs are climbing, QA needs to improve. Also, trivial user-reported issues might not be critical. So focus on significant bugs. This KPI encourages thorough testing and robust test planning. However, note that a 100% internal detection is unrealistic in complex systems; the aim is to minimize high-severity escapes. QA can track “escaped bugs” each release and do a quick post-mortem on why they were missed to improve test coverage next time.
* **Test Coverage (Requirements coverage):** *Definition:* The percentage of requirements or user stories for which test cases have been written and executed. Alternatively, could measure coverage in terms of code (if automated tests, though QA primarily covers functional coverage). But better for manual QA is requirement coverage: ensure each story’s acceptance criteria are fully tested. *Target:* 100% of user stories tested (no feature goes out untested). And if using traceability, each requirement has at least one test case linked. *Anti-gaming:* It’s possible to have test cases but of poor quality. So combine with bug detection: if coverage is “100%” yet many bugs escape, maybe tests weren’t effective. Still, this KPI ensures QA reviews all features. For automated tests, code coverage % can be a supporting metric (like aiming for, say, >70% unit test coverage if QA also handles automated tests with dev). QA works with dev to ensure critical code paths have some form of test.
* **Test Cycle Time:** *Definition:* Time taken to test a given release or feature cycle. For example, “time from code freeze to end of testing sign-off”. Or how long it takes QA to fully test a story after dev completes it. *Target:* Efficient turnarounds - e.g., for a two-week sprint, final regression testing within 1-2 days; for a user story, maybe tested within 1 day of dev completion. *Anti-gaming:* This should not push QA to rush and miss things; rather, it encourages better automation and test process to speed up. If testing is taking too long (slowing releases), analyze why: maybe environment issues or insufficient automation. With improved automation or parallel testing, cycle time should drop. The KPI can be measured over iterations to see if testing phase is shrinking or at least not ballooning as product complexity grows. It signals whether QA can keep up with dev output. If dev is faster than QA can verify, that’s a bottleneck to address (via automation or more QA resources).
* **Defect Turnaround Time:** *Definition:* The time from when a defect is logged by QA to when it’s verified fixed and closed. Essentially, how quickly are bugs being resolved. QA influences this by prompt retesting and clear bug reporting. *Target:* On average, high-priority bugs fixed and closed within, say, 1-2 days; lower priority maybe within the release cycle. *Anti-gaming:* QA cannot directly control dev speed, but by making bug reports actionable and by quickly re-verifying after a fix, they prevent delays. If this time is high, it could indicate either slow dev response or too many back-and-forths due to unclear reports. So QA uses this to improve clarity and prioritize well. A good collaboration yields quick fix/verify cycles. Perhaps measure “percentage of bugs reopened” too - ideally low, meaning fixes solve the problem first try (indirectly reflecting good communication of bug details). A low reopen rate (say <5%) suggests QA described issues well and devs fixed correctly.
* **Automation Coverage & Effectiveness:** *Definition:* If test automation is part of QA’s scope, measure how many of the regression test cases are automated, or how long the manual regression takes now vs before automation. Also track if automated tests are catching bugs (e.g., number of failures that revealed real bugs rather than flakiness). *Target:* Increase automation coverage gradually (e.g. automate 5 new test cases each sprint) and reduce manual regression time by X%. Also, aim for low false-fail rate in automated tests (flaky tests under 2% of runs). *Anti-gaming:* It’s not just quantity of tests, but ensuring they’re meaningful and maintained. If automated tests exist but always failing (flaky) or being skipped, coverage number is useless. So measure stable, passing coverage that contributes to quality. QA can use this to justify time spent on automation (e.g., “We automated the login tests which run on each build, catching 3 issues already, and saved us repetitive work”). Automation effectiveness might be anecdotal but track the trend of manual hours saved. This metric ensures QA effort is partly invested in future efficiency, not only in manual testing each cycle.

**Career Levels & Growth:** A **QA Engineer** in a startup might start as **QA Analyst/Tester (junior)** focusing on executing tests, then become a **QA Engineer/Senior QA** who designs test strategies, leads testing for major releases, and mentors others. They could grow into a **QA Lead** or **QA Manager** role as the team expands (managing other QA engineers, interfacing with management on quality initiatives). Another path is towards **Automation Architect or SDET (Software Development Engineer in Test)**, focusing deeply on building test frameworks and tooling if they have coding inclination. In some cases, experienced QAs move into **Product roles** or **User advocacy roles** since they have a broad view of the product. For growth in QA, signals include: being proactive in improving quality processes, effectively handling more complex projects (e.g., being the sole QA on a big feature and it goes out with minimal issues), and taking initiative to introduce automation or better tools. A senior QA often influences requirements too (bringing in the quality/testability perspective early). As levels increase, a QA may also become responsible for performance testing, security testing coordination, or overall release management aspects. Eventually, a **QA Manager/Lead** might set up entire QA strategies across multiple teams (if the startup has several squads) and coordinate UAT with external stakeholders, etc. They might also own metrics like overall product quality KPIs, and training junior QAs or devs in testing.

***Signals of excellence:*** The hallmark of a top-notch QA is that major issues rarely reach customers - they catch them. Releases under their watch are smooth, instilling confidence in the team that “if QA says it’s good, it’s good.” They have an almost intuition for where to find bugs - colleagues might comment that this QA “thinks of all the edge cases.” They also balance speed and thoroughness expertly: they know what to test exhaustively and what to skim (risk-based testing), so they aren’t a bottleneck but also don’t let critical things slip. Their bug reports are clear and action-able; developers thank them for the detail which made fixes easier. Over time, an excellent QA also helps raise developers’ quality awareness - you’ll notice devs start to ask themselves “what would QA test here?” and include that in their own work, meaning fewer bugs introduced. The QA likely spearheaded automation that runs in CI, so regression is caught within minutes of code changes. They keep improving testing methods - for instance, introducing API testing that shortened UI test times or using data generation tools to broaden test coverage. They may also contribute to product improvements by reporting not just bugs but UX issues or potential performance problems early. Feedback from PMs and devs would be that this QA is reliable, detail-oriented, and a strong communicator. Quantitatively, you might see that under this QA, the bug escape rate to production went down release over release, or customer-reported issues dropped. Also, an excellent QA manages their time well - they rarely say “I didn’t get to test that at all” because they plan and communicate constraints early to allow adjustments. They are proactive: if a release is in jeopardy due to quality, they sound the alarm with solutions (maybe they propose a release delay or a scope cut for quality sake, and they’re respected enough to be heeded). Another signal: they maintain a well-organized repository of test cases and reports, so anyone can see what was tested and what the results were. Essentially, excellence in QA is seen in product stability and team trust.

***Failure modes:*** A QA can fail by missing critical bugs that impact customers (everyone will eventually miss something, but a pattern of missing obvious issues is problematic). If many bugs slip through that a reasonable testing should catch, that QA might be too cursory or lacking good test strategies. On the flip side, a QA who **blocks releases unnecessarily** can also be failing - e.g. raising trivial issues as showstoppers or not aligning their rigor with business needs. Being too adversarial or nitpicky can strain dev relations and slow the team with little benefit. Another failure sign is disorganization - e.g., they don’t have test cases written down, so testing is ad-hoc and inconsistent; this leads to things being forgotten or tested differently each time. Or if they can’t provide a clear answer on what’s been tested vs not, leaving uncertainty. Also, **poor bug reporting** can hamper the team: if a QA reports bugs without steps or with insufficient info, devs waste time or ignore the bug until clarified, delaying fixes. If a QA takes feedback personally or becomes combative with devs (“It’s not my job to think of that” or vice versa), that’s toxic - they must remember it’s a team effort, not QA vs dev. Being slow or unresponsive is another issue: if devs are waiting for QA to verify fixes or test a build and QA is not keeping up, it causes frustration and might tempt devs to merge code without proper QA (dangerous). Also, a QA who doesn’t update their testing approach as the product grows (like still doing only manual tests when automation is needed, or not learning how to test new tech like a new API or mobile interface) can become a bottleneck by being out of depth. Essentially, if quality of releases is consistently poor and traceable to testing gaps, or if the QA’s interaction with the team is hindering progress (either by delaying or by conflict), then the QA is not fulfilling their role well. These scenarios call for additional training, better process, or a discussion on expectations.

**Weekly Action Plan for QA:**

* **Sprint Planning/Start:** As the sprint begins, review all the user stories and requirements slated. Begin identifying test scenarios for each story (you might even write some test cases up front if things are clear). If any acceptance criteria are vague or missing, now is the time to ask the BA/PM or dev for clarification - don’t wait until development is done. Note any special test data or environment needs and arrange them. For example, if one story involves testing an email sending function, ensure you have access to a test email account or SMTP logs. If you foresee a need for any new tool or access (like “we might need to monitor database changes for this feature”), communicate that early so dev/ops can help set it up. Basically, get prepared to test *in parallel* with development. You can also prioritize - mark which features are high risk and will need more thorough testing or early testing. Share your testing plan outline with the team if possible (it shows preparedness and might trigger devs to mention edge cases they know).
* **During Development:** Stay engaged with daily stand-ups or syncs. If a developer finishes a feature early, be ready to test it as soon as a build is available - *don’t* wait until the end of the sprint to test everything. Testing in small chunks helps find issues while the dev still has context fresh to fix them. So, try to test stories as they are completed. Communicate actively: if a dev says “Feature A is done and on dev environment,” jump on it and test within a day. If you find a critical bug, inform the dev immediately (file it and perhaps ping them). If no features are ready on a given day, you can use time to work on automation or test planning for upcoming items, or do exploratory testing on previous features. Keep updating your test cases as requirements evolve or as you discover new scenarios to test. Also, do quick regression checks around any changed area - e.g., if dev updated the login logic, quickly verify login, logout, and maybe sign-up still work. Catch side effects early. Additionally, keep an eye on unit test results (if CI runs them) - sometimes they indicate potential issues that QA should be aware of (“I see a lot of tests failing around payment calculation, maybe there’s a bigger logic error dev is working on - I’ll note to double-check that thoroughly when ready”).
* **Mid-sprint Checkpoint:** By mid-sprint, ideally a few features have been tested or at least test cases prepared. Let’s say half the stories are done - make sure you’ve tested those or at least done a smoke test. Update the team in stand-up about testing status: e.g., *“Tested 3 out of 5 new features, found 2 major bugs (already reported), waiting on fix for feature X”*. If you foresee a crunch - e.g., if many features will only be ready at the very end - flag that risk: *“It looks like 4 stories will be completed on the last day; I might need a day or two extra to test all thoroughly or we reduce scope”*. This transparency might prompt devs to finish a bit earlier or help out with testing. Check on any blocking issues: if you’re waiting on a fix or an environment setup, remind the responsible person politely. Also, take some time mid-sprint for **exploratory testing** on the integrated build - features that are done, try using them in combination or in unpredictable ways. This might catch integration issues (like Story A and Story B individually work but together cause a problem). Use exploratory charters - e.g., “Test user journey from sign-up to creating a project to logging out” - see if the whole flow holds up. Share any serious findings. Also mid-sprint, you might do **test case maintenance**: update any cases that became obsolete or add new ones for discovered edge cases.
* **End of Sprint & Regression:** As the sprint concludes, ensure you do a round of **regression testing** for core functionalities. Even if not explicitly asked each time, it’s a good practice to quickly verify that the previously working key features (login, main transactions, etc.) still work after new changes. If you have automated regression, run it and review results. If manual, have a short checklist of “smoke test” items you always go through pre-release. This catches if something unrelated broke. Also, verify all fixed bugs: any bug marked resolved should be re-tested on the latest build and closed if indeed fixed. During the **sprint review/demo**, listen carefully - sometimes while a feature is demoed by a dev or PM, you might notice an issue that wasn’t caught. If it’s minor, note it to file after; if major, possibly mention it if appropriate (“I noticed on demo the app crashed when X, I will investigate that”). In the **retro**, be the quality advocate: mention if testing went well or not. For example, *“We had too many last-minute merges, making it hard to test thoroughly - maybe next time, cut off dev a day earlier.”* Or *“The early testing of feature Y helped catch a design issue, which was great - let’s do that always.”* Also express any needs: *“Our automated tests caught a bug this time, that saved a lot of time - we should invest more in automation.”* Or if something like environment downtime hindered you, suggest a solution. The retro is your chance to subtly educate the team on how to improve quality process, without blaming, focusing on solutions.
* **Ongoing: Bug Triage and Automation:** Make bug triage a continuous process. Every time you log bugs, ensure they are prioritized (if not by PM, then at least tag severity for dev/lead awareness). Offer to help dev/PM in prioritizing if needed by explaining impact. Keep the bug list tidy: close duplicates, update statuses if a dev forgot to, etc. This administrivia keeps everyone clear. For automation, try to allocate some time each sprint to maintain or add to the automated test suite. Perhaps aim to automate one new high-value test case per sprint. If it’s hard to find time during main work, see if during a slower day or after a release you can focus on it. Automated tests will ease future weeks. Also ensure existing automated tests stay green - if any are failing, identify if it’s a product bug (report it) or a test script issue (fix it). Don’t let the suite become ignored due to false alarms.
* **Build a Knowledge Base:** Over weeks, you likely gather lots of details about the product’s quirks and edge cases. Maintain documentation of test cases or at least a checklist that can be reused for regression. Note down environments and credentials for testing, any special setups (like “to test payment, use credit card number X given by Stripe docs”). Also maintain a record of past bugs and how they were fixed - this is useful if something resurfaces or to remember to check related areas. When new QAs or devs join, these notes become very helpful to onboard them about tricky parts of the system. In a startup, formal documentation is often lacking, so your QA test case repository might double as part of product documentation, capturing expected behaviors. Curate it well (maybe on Confluence or wherever accessible).
* **Coordination before Release:** If your development is sprint-based but releases are less frequent, you might have an extra pre-release phase. As QA, coordinate with PM and dev when a release is code frozen for final testing. Intensify regression tests, performance tests, etc., during that period. Also, if doing a release candidate build, run through a **release checklist** - e.g., correct version number, new features toggled correctly, migrations run in test environment, etc. Communicate a clear “QA sign-off” or list of known issues to decision makers. If you aren’t confident in the release, articulate the risks clearly: *“We have not tested X scenario due to time, so there’s a risk there; also bug #123 is unresolved which could affect 5% of users - I recommend fixing that first or warning support.”* Provide options (delay, do a patch, etc.) rather than just “It’s bad.” Then abide by whatever decision is taken, but at least you did due diligence. Once a release goes out, perhaps do a quick smoke test in production (if feasible) to ensure everything deployed correctly. Monitor if any alerts or user feedback come immediately and be ready to respond. Then breathe - and prepare for the next cycle, incorporating lessons learned.

## Reusable KPI Library

To complement the role-specific KPIs above, here is a **library of common engineering KPIs** that early-stage startups can use or adapt. These metrics help quantify different dimensions of software team performance and product health. They are organized by category (speed, quality, etc.), with typical definitions and notes:

| **KPI** | **What it Measures** | **Formula/Definition** | **Cadence** | **Notes (Anti-gaming)** |
| --- | --- | --- | --- | --- |
| **Sprint Velocity** (Throughput) | Team output per sprint in terms of completed work. | Sum of story points (or count of stories) completed in the sprint. | Per sprint. | Use as a planning tool, not a target to inflate. Velocity will stabilize over time; focus on consistency over raw high numbers. |
| **Sprint Predictability** (Commitment Reliability) | How well the team meets its sprint commitments. | (Completed SP / Committed SP) \* 100%. E.g., 85% means most planned work was done. | Per sprint. | Don’t game by under-committing; aim for realistic commitments. A consistently low ratio suggests overcommitment[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear). |
| **Lead Time for Changes** | Speed from code commit to code in production (or to users). | Time difference between a code change being made and that change deployed live. Often measured as median hours/days. | Continuous (tracked via CI/CD pipeline data). | Shorter lead time means faster delivery of value. Watch out: very short lead time but high failure rate is not good - balance with quality metrics. |
| **Deployment Frequency** | How often the team deploys to production. | Number of production deployments per day/week/month (depending on frequency). | Weekly or monthly reporting. | Higher frequency means smaller, iterative releases. Ensure deployments are successful; don’t split one change into many tiny deploys just to boost count. |
| **Change Failure Rate** | Quality of releases - what percent of deployments cause issues. | (# of deployments that resulted in rollback or hotfix / total deployments) \* 100%. (DORA metric) | Per release or monthly. | A lower rate is better (few failures). If high, focus on testing, code review, and incremental releases. Don’t hide failures - track them to improve. |
| **Mean Time to Recover** (MTTR) | How quickly the team fixes incidents or outages. | Average time from incident start to resolution (restoring service). | Per incident (average over period). | Lower MTTR means resilient operations. Improve by clear on-call processes. Don’t game by marking incidents resolved before they truly are - measure until full recovery. |
| **Automated Test Coverage** | Portion of code or functionalities covered by automated tests. | Could be % of lines of code covered by unit/integration tests, or % of test cases automated. | Each code merge (for unit test), or monthly for overall. | Higher coverage can prevent regressions. But 100% coverage isn’t a goal if tests are low quality. Don’t write trivial tests just to hit a number - focus on critical logic coverage. |
| **Escaped Defects** (Post-release bugs) | Bugs users find that were not caught internally. | Count of valid bug reports from users in a release or time period (maybe weighted by severity). | Per release or monthly. | Trend should be downward - as internal QA improves, fewer escapes. If rising, strengthen testing in those areas. Encourage users to report - an initial uptick might just mean better feedback, not worse quality. |
| **Customer Reported Issues Fixed** | Team responsiveness to user-reported problems. | % of user-reported issues resolved within a SLA (e.g., 1 week). Or simply count fixed vs reported. | Monthly. | Shows attentiveness to customer feedback. Avoid gaming by not counting certain reports - define what’s tracked (e.g., production bugs, not enhancement requests). High percentage of quick fixes means good support. |
| **Application Crash Rate** | Stability of the application. | Crashes per 1,000 sessions (for an app) or error rate per 1,000 requests (for a service). | Daily/Weekly (tracked via monitoring). | Aim to minimize. A spike indicates a serious regression. Ensure you have monitoring in place. Don’t suppress error logging to fake a lower rate - address root causes. |
| **Page Load Time / Latency** | Performance experienced by users. | For web: average page load time (e.g. homepage = 2.3s). For backend: API response time (e.g. p95 = 120ms). | Continuous (monitoring dashboards). | Improves UX; track over time especially after releases. Focus on 95th percentile (p95) or p99, not just average, to catch tail latency. Don’t over-optimize something that’s already sub-second unless needed; prioritize pages that matter most to users. |
| **Uptime** (Availability) | How much time the service is up and reachable. | Measured as a percentage (e.g., 99.8% uptime per quarter). Uses monitoring pings or service health checks. | Daily aggregated, reported monthly/quarterly. | High uptime is crucial for user trust. Ensure maintenance windows are accounted or excluded if applicable. Don’t cheat by not reporting incidents - be honest to drive improvements. |
| **Team Happiness / Morale** (SPACE “Satisfaction”) | Team’s self-reported morale and perceived productivity. | Collected via periodic anonymous surveys (e.g., eNPS or simple 1-5 rating). | Quarterly. | Health of the team. A happy team is more productive and creative. Watch trends; a dip might predict burnout or issues. Not for gaming, but for discussion - encourage candor. |
| **Knowledge Sharing / Collaboration** | How much the team is collaborating and spreading knowledge. | Could be # of peer code reviews done, # of lunch-and-learn sessions, or rotation of on-call duty among members. | Monthly/Quarterly. | Qualitative but important. e.g., 100% of PRs have >=1 peer reviewer ensures collaboration in code. Don’t just count sessions - ensure they’re effective (get feedback). |

This library provides a menu of metrics - an early-stage startup should pick a handful that align with current goals (e.g., if speed of delivery is key, focus on lead time and deploy freq; if quality is suffering, focus on escape rate and failure rate; if team is growing rapidly, maybe team happiness and collaboration metrics matter to maintain culture). **Remember: KPIs are tools, not ends in themselves.** Use them to identify areas to improve, not to blame. And always supplement numbers with context[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear) - e.g., an uptick in escaped bugs might be acceptable if it came with doubling the feature delivery (maybe a conscious trade-off). The key is to have *balanced metrics* across productivity, quality, and morale (like the SPACE framework suggests) to avoid sub-optimizing one area at the expense of others.

## Case Studies: Adapting Roles & KPIs in Practice

Let’s look at two hypothetical startup scenarios to see how clarifying roles and adjusting KPIs made a real difference.

**Case Study 1: Improving Quality by Defining the QA Role at FinTechCo**  
FinTechCo (50 employees) found that every product release was plagued with customer-reported bugs, hurting its credibility. Developers were each doing a bit of testing, but there was no dedicated QA ownership - some critical flows slipped through untested. Customers were discovering issues like calculation errors in reports and mobile app crashes. Over one quarter, FinTechCo appointed a **QA Lead** and defined a formal QA Engineer role with clear mission and responsibilities. They introduced a checklist that *no story is done until QA signs off*[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear). The QA Lead set up basic KPIs: **escaped bugs per release** and **test coverage of user stories**, to track improvements. Initially, 5 major bugs escaped in the June release. The QA Lead implemented thorough test case design for each user story and a smoke test suite for regression. By the September release, escaped bugs dropped to 1 minor issue. Developers reported that having QA involved early actually raised their own quality awareness; they started running unit tests and doing sanity checks before handing to QA, because they had clear criteria to meet. This aligns with industry findings that incorporating quality earlier leads to dramatically fewer production issues.

**Case Study 2: Pivoting Developer KPIs to Drive Team Health at ShopEase**  
ShopEase, an e-commerce startup (120 people), was scaling quickly and pushing the dev team hard to release new features weekly. They had set aggressive KPIs like *features per quarter* and *lines of code written*, hoping to motivate output. But developers started burning out; quality suffered and morale dropped. Recognizing the issue, ShopEase’s VP Engineering pivoted to a more balanced set of KPIs inspired by the **DORA and SPACE frameworks**. They scrapped “lines of code” (which was encouraging bloat) and introduced **Lead Time** and **Change Failure Rate** to measure delivery speed *with* stability. They also added a **Team Satisfaction** score via quarterly surveys. Responsibilities of the **Dev Lead** role were clarified to include mentoring and ensuring sustainable pace, not just output. Over the next 2 quarters, they found that while deploy frequency stayed constant, the change failure rate (bugs causing rollback) dropped from 15% to 5% as developers focused on quality and pair programming more. Interestingly, the Team Satisfaction score rose from 3.1 to 4.0 (out of 5) in the same period - developers felt heard and less like code machines. A senior developer said, *“We started measuring what actually matters - not how much code we churn, but how quickly and reliably we deliver value. And knowing leadership cared about our well-being (via the survey KPI) made a huge difference.”* ShopEase’s lesson: aligning KPIs with *both* performance and people (e.g., DORA metrics + morale metrics) created a healthier, more productive team. The improved stability also meant less time firefighting, more time building - a true win-win in practice.

## Weekly Rollout Plan for JDs & KPIs Initiative

Implementing new job definitions and KPIs in a growing startup needs to be done methodically to ensure buy-in and lasting impact. Here’s a week-by-week rollout plan **(over ~8 weeks)** that an engineering leadership team could follow to put in place the structured JDs and KPIs described in this chapter:

**Week 1: Initiation & Team Buy-In**  
- *Kick-off Meeting:* Announce the initiative to the engineering and product teams. Explain why clearly defined roles and KPIs matter - use some points from the Introduction (e.g., how it improves focus, transparency, and performance). Emphasize this is to help the team, not to micro-manage.  
- *Distribute Draft Role Definitions:* Share the drafted job descriptions for TPM, BA, Dev Lead, Tech Lead, Dev, QA, Test Automation (perhaps as a document or Confluence page). Allow team members to review and digest.  
- *Gather Initial Feedback:* Set up a Slack channel or anonymous form for feedback/questions on the role definitions. People might be unsure how these differ from current practices - encourage them to ask. For example, a developer might ask “Will the Dev Lead still code or only coordinate?” - great opportunity to clarify (Dev Leads code ~50% and also guide the team).  
- *Identify Champions:* Informally identify one team member excited about each role definition or KPI aspect. For instance, maybe a senior dev who cares about code quality could champion the KPI effort for quality metrics. These champions can help advocate to others.

**Week 2: Refine Roles & Responsibilities**  
- *Team Workshops:* Hold short workshops (1-2 hours) for each functional role group (one with all devs, one with product folks like BAs/PMs, one with QA, etc.). In each, walk through the relevant role template. e.g., with devs, review the Dev Lead, Tech Lead, and Developer definitions. Encourage open discussion: “Does this cover your current pain points? Anything unclear or missing?”.  
- *Finalize JD Documents:* Incorporate feedback from workshops. Maybe devs agreed that the Tech Lead should also own performance benchmarking (add that), or QA suggested the “failure modes” section include missing test environments. Keep the core, but adjust wording for the team’s context. Have managers approve the final version of each JD.  
- *One-on-One Check-ins:* Managers meet 1:1 with individuals particularly impacted by role changes. For example, if someone has been acting as both Dev Lead and Tech Lead, discuss how duties might shift. Ensure everyone knows *which* role they are filling moving forward, especially if people had multiple hats. This prevents ambiguity or turf issues when the new definitions go live.

**Week 3: KPI Selection & Definition**  
- *Leadership KPI Brainstorm:* Engineering managers, Tech Lead, PM, etc., meet to pick a balanced set of KPIs to pilot (maybe 4-6 total). Use the KPI Library as reference. Decide which **team performance** metrics (like sprint predictability[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear), lead time), **quality** metrics (escape rate, change failure rate), and **team health** metrics (satisfaction) make sense for your context. Ensure each maps to a goal (e.g., we choose “Deploy Frequency” because we want faster iterations).  
- *Define KPI Operational Details:* For each chosen KPI, define exactly how to measure it (who collects data, from where) and the target or baseline. E.g., “Deployment Frequency: currently 1/week, target 2/week by next quarter; measured via CI pipeline stats.” Document these clearly.  
- *Tooling Prep:* Set up any tools needed to track these KPIs. This could mean configuring Jira for reporting sprint commitment vs completed, adding monitoring for uptime or response times, setting up a team satisfaction survey in a tool like CultureAmp or Google Forms. If a metric is new, maybe a script or logging needs to be implemented (like capturing lead time from commit to deploy - might pull from Git and CI data). Plan this out now.  
- *Communicate KPI Plan:* Share the list of KPIs with the team along with the “why” of each (reference industry data or DORA/Atlassian research as rationale[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear)). Emphasize these are *team* KPIs, not individual metrics, and not for blame - they are to identify improvements. This managing of perception is crucial so people support it. Possibly get input: “Are these metrics understandable and do we foresee any issues gathering them?” Tweak if a strong case is made (e.g., maybe lines of code was mistakenly included and a dev points out it’s a poor measure - agree and drop it).

**Week 4: Pilot and Dry Run**  
- *Implement Role Changes in Real Work:* This week, act according to the new JDs and see how it goes. For example, let the TPM actually run the sprint planning meeting (if previously a dev lead did) - give them that ownership as per the TPM responsibilities. Have the QA lead start enforcing the “QA sign-off” rule for done. Essentially, each person practices their “contracts” as defined. Managers should remind and reinforce: e.g., ping the Tech Lead to do that architecture review they are supposed to do, etc.  
- *KPI Collection Dry Run:* Without making any big deal of numbers, quietly collect data for the KPIs this week to establish a baseline. E.g., measure the current sprint’s predictability (maybe it was 60%), current deployment freq, etc. Also send out the team satisfaction pulse survey (mark it baseline). Share the baseline results with the team in a retro or meeting, discussing neutrally. For instance: *“We delivered 50% of planned stories - seems we consistently over-plan. Let’s calibrate next sprint.”* Or *“Our lead time from merge to deploy is 4 days; maybe we can automate testing to cut that.”* This conversation gets people used to metrics as learning tools.  
- *Address Role Issues:* Check in mid-week - did any role confusion occur? Maybe two people still both tried to do a task (old habits). Talk it through in real time. E.g., a Dev Lead might still be doing code reviews that the Tech Lead should do - clarify division again. By end of week, hold a brief retro specifically on “How is the new role clarity working for everyone? Any blockers or concerns?” This gives the team ownership - they might have great suggestions to further refine responsibilities or collaboration protocols (like daily stand-up format changes now that TPM leads it).

**Week 5: Rollout Adjustments & Formalization**  
- *Tweak and Formalize JDs:* Incorporate any fixes from the pilot week into the job description docs. Now save/publish them in an official place (in HR system or wiki) and consider them the new standard. Announce that as of now these roles and responsibilities are in effect. (If any reporting line or title changes needed to accompany, handle that with HR).  
- *Begin Official KPI Tracking:* Set up a visible dashboard or scoreboard for the chosen KPIs. For example, a Confluence page or a wallboard that shows the metrics each sprint or month. Populate it with last week’s baseline and this week’s data. Don’t set strict targets yet if unsure; maybe just track initially. You might, however, set a team goal like “improve sprint predictability by 20% in next 2 sprints” or “reduce escaped bugs to <2 per release” to give a sense of direction. Make sure the team knows where to see the metrics. Possibly integrate into existing meetings: e.g., mention KPI status briefly in each sprint review (“By the way, our deploy freq this sprint was 2, up from 1 last sprint - nice!” or “We had 1 escaped bug, let’s aim for zero next time”). This regular mention normalizes the metrics[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear).  
- *One-on-One Coaching:* Managers meet any individuals who still seem unclear or slightly resistant to changes. For instance, if a senior dev not in a lead role feels demotivated by focus on metrics, reassure how it’s for team improvement not personal evaluation. Or coach a new Dev Lead on how to balance coding vs coordinating now that it’s formal. Provide support - maybe pair them with a mentor or allocate some management time to guide them initially in their new duties.

**Week 6: Monitor & Stabilize**  
- *Routine Operations:* By now, each role should be routinely executing their parts: TPM handling ceremonies and stakeholder comm, Tech Lead doing design reviews, QA running the test suites, etc., as per JDs. The engineering manager/CTO should observe and ensure no major gaps - basically acting as a safety net until new habits form. Reward or call out successes, e.g., “Great backlog grooming session run by our BA today - exactly what we needed for clarity.” Positive reinforcement encourages everyone to take pride in their defined roles.  
- *Track KPI Trends:* After a couple of data points, gather the team to quickly look at trends. *“Velocity is stabilizing around 20, up from 15 a month ago - good sign of throughput increase.”* Or *“Our customer bug reports have dropped to 1 this week, down from 5/week early last month.”* Connect improvements to actions: maybe fewer escaped bugs is thanks to the QA process; celebrate that cause-effect to show it’s working. If any metric is moving the wrong way, discuss openly in retrospective: *“Our deployment frequency fell, possibly because our build pipeline slowed down - let’s investigate improvements there.”* Team problem-solving around metrics will increase buy-in.  
- *Prevent Slippage:* It’s easy after initial excitement to revert to old habits. Watch for any slide: e.g., a Dev Lead skipping one-on-ones or a stand-up going long because TPM didn’t timebox. Gently remind and reset as needed. It might help to create a role-responsibility RACI chart visible to all, so if confusion arises, one can quickly check (e.g., “Tech Lead: Accountable for code architecture; Dev Lead: Responsible for task assignment & delivery tracking”). This can be a reference that team members can point to if boundaries blur.

**Week 7: Scale & Integrate**  
- *Integration with HR/Org:* Work with HR to integrate these JDs into career frameworks or job listings. If hiring, ensure new postings reflect these clear roles (that way new hires come in knowing what to expect). Internally, you might update titles if needed (maybe some “Project Managers” are now called “Technical Program Managers” officially, etc., to match definitions). If performance review season is near, plan to use the role definitions as a basis for evaluation - e.g., review how well someone fulfilled their listed responsibilities and signals. This cements the importance of the JDs.  
- *Cross-team Alignment:* If your startup has multiple teams or functions, share what you’ve done. Perhaps product or design teams will clarify their roles in parallel, or at least be aware that engineering now has a TPM - they should know to funnel certain communications through that TPM for efficiency. Align with other departments if needed, e.g., let Customer Support know QA is now doing root cause analysis for bugs so they can route tech queries differently. Smooth out inter-team processes with the new roles in mind (for example, sales requests for dev team go via BA or PM now, not ad-hoc to any dev).

**Week 8: Review & Future Improvements**  
- *Post-Mortem of Initiative:* Conduct a full retrospective on the rollout itself with the engineering team (and relevant others). What went well? (Maybe “defining roles eliminated confusion and things aren’t falling through cracks now”). What didn’t? (Maybe “some KPIs we chose aren’t actually easy to measure or weren’t that useful”). Use this to refine. For instance, drop or replace a KPI if it’s not serving value. Or adjust a responsibility if experience shows a tweak is needed. This is continuous improvement - the chapter’s practices can evolve with your startup.  
- *Set Next Quarter Goals:* Leverage the momentum by setting some concrete goals now that baseline metrics are established. e.g., “Q2 Objective: Achieve Elite performance in DORA metrics - deployment >1/day, lead time <1 day, change fail <5%, MTTR <1h.” Or a goal like “Zero P1 bugs escape to production for 3 months straight.” Tie these goals to business outcomes (faster shipping, better user retention through quality) to keep everyone motivated on *why* these roles and KPIs matter.  
- *Celebrate Wins:* Finally, acknowledge how far the team has come. If sprint predictability went from 50% to 80%, or customer NPS improved due in part to better quality, call that out and thank the team for adapting to the changes. Consider a small team event or shout-out. People will embrace processes that they feel improve their work-life and product success - and celebrating helps reinforce that positive association.

By following a rollout plan like this, a startup can systematically embed the job descriptions and KPI-driven culture into its DNA. The key throughout is **communication, adaptation, and reinforcement**. Within 2 months, the organization should feel more structured yet still agile: everyone knows their role in the “race team” and the KPIs are the dashboard helping steer to victory. As one Atlassian case study famously implied, when teams have clarity and measure what matters, they can achieve high performance with a healthy culture - exactly what early-stage startups need to scale successfully.

## References

1. Forsgren, N. et al. (2021). *“The SPACE of Developer Productivity: There's more to it than you think.”* ACM Queue. (Research from GitHub & Microsoft highlighting the need for multiple metrics - Satisfaction, Performance, Activity, Communication, Efficiency - to truly gauge dev productivity rather than a single number).
2. Google Cloud (2023). *“Announcing the 2023 State of DevOps Report.”* (DORA research key findings: high software delivery performance correlates with both higher org performance and better employee well-being. Defines metrics like Lead Time, Deployment Frequency, Change Fail Rate, MTTR as critical benchmarks).
3. Atlassian Work Life Blog - Agnes Ro (2016). *“When I grow up, I want to be… a development lead?”* (First-person account of transitioning to a dev team lead at Atlassian, emphasizing people management, balancing product vs tech vs quality demands, and the rewarding challenge of leadership).
4. Atlassian Team Playbook. *“KPIs: Understanding Metrics for Business Success.”* (Guide on choosing and using KPIs effectively - stresses alignment with business objectives, making KPIs specific and actionable, and avoiding vanity metrics)[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear).
5. Accelerate (2018) by Nicole Forsgren, Jez Humble, Gene Kim. (Pioneering book on DevOps and performance - provides evidence that implementing DevOps best practices and measuring the four key metrics greatly improves software delivery and organizational performance. Basis for DORA metrics used in this chapter).
6. J. Humble & D. Farley (2010). *“Continuous Delivery.”* (Book that popularized deployment frequency and cycle time as vital metrics; many startups have drawn from its principles to shorten lead times and iterate faster safely).
7. Martijn Verbannen (Dev.to, 2020). *“A Guide to Effective One-on-Ones for Engineers.”* (Resource for Dev Leads/Engineering Managers on coaching team members - relevant as we instituted more defined lead roles and their duty to mentor and support developers)[[4]](https://www.geeksforgeeks.org/hr/team-lead-vs-tech-lead-difference-between-team-lead-and-tech-lead/#:~:text=,final%20assiduity%20trends%20and%20technologies).
8. Scrum Guides - Schwaber & Sutherland (2020). *“The Definitive Guide to Scrum.”* (Defines roles like Product Owner, Scrum Master (TPM analog), Dev Team responsibilities. Underpins our definition of TPM and how they facilitate agile ceremonies and remove impediments).
9. DORA DevOps Quick Check (2021). *(Online assessment tool by Google’s DevOps Research & Assessment - helps teams measure their performance on key metrics. Reinforces the idea of using metrics to identify bottlenecks and improve incrementally)*.
10. **Company X Case Study** (Fictional reference combining common real startup experiences drawn from the author’s knowledge). Illustrates scenario outcomes like bug reduction and morale improvement when implementing JD & KPI changes.

[[1]](https://www.geeksforgeeks.org/hr/team-lead-vs-tech-lead-difference-between-team-lead-and-tech-lead/" \l ":~:text=Functionalities%20of%20a%20Tech%20Lead,can%20include) [[2]](https://www.geeksforgeeks.org/hr/team-lead-vs-tech-lead-difference-between-team-lead-and-tech-lead/#:~:text=,arise%20during%20the%20development%20process) [[4]](https://www.geeksforgeeks.org/hr/team-lead-vs-tech-lead-difference-between-team-lead-and-tech-lead/#:~:text=,final%20assiduity%20trends%20and%20technologies) Team Lead vs. Tech Lead | Difference Between Team Lead and Tech Lead - GeeksforGeeks

<https://www.geeksforgeeks.org/hr/team-lead-vs-tech-lead-difference-between-team-lead-and-tech-lead/>

[[3]](https://www.atlassian.com/work-management/project-management/project-planning/kpi#:~:text=%2A%20Strategic%20planning%20and%20decision,Regular%20KPI%20tracking%20establishes%20clear) KPIs: Understanding Metrics for Business Success | Team Central

<https://www.atlassian.com/work-management/project-management/project-planning/kpi>