GitHub's Engineering System Success Playbook

Contents

Introduction: GitHub engineering zones and metrics

At GitHub, we know that better business outcomes aren't driven just by good-quality code, speed, or developer happiness in isolation. It's actually when quality, velocity, and developer happiness are working in unison that organizations see their best results. If you're looking for engineering to provide greater value to your business, it's crucial to strengthen these — let's call them — foundational zones, and create better conditions for your teams to thrive.

This is the crux of GitHub's Engineering System Success Playbook (ESSP) — a three-step process that can help you drive meaningful, measurable improvements in your organization, whether you're looking to adopt a new AI tool like GitHub Copilot or identify and unlock bottlenecks that have been hindering performance.

Inspired by multiple frameworks, including SPACE and DevEx, DX Core 4, and DORA, our playbook offers a balanced and comprehensive approach, helping you assign metrics to each "zone" that you can track over time and iterate as needed.

> **Here's a quick breakdown of the process:**
>
> **Step 1:** Identify the current barriers to success
>
> **Step 2:** Evaluate what needs to be done to achieve your goals
>
> **Step 3:** Implement your changes, monitor results, and adjust

At the heart of our ESSP is a systems thinking[1] approach that prioritizes long-term, sustainable improvements. While quick wins can be a great way to get an initiative started, they can produce negative downstream effects. For example, accelerating code review turnaround time can speed up development, but without addressing the broader system – like testing infrastructure and documentation practices – you may risk creating bottlenecks downstream and compromising code quality.

This project was created in response to many customer requests for prescriptive guidance on creating meaningful downstream impact from changes in their engineering systems—

1: A *system* is a group of interrelated, or interdependent parts that together serve a function or purpose ('Thinking in Systems' by Donalla Meadows). *Systems thinking* brings a focus to the *relationship* between the multiple parts in the system (The Systems Thinker), recognising that the whole has emergent properties that are different to the sum of its parts.

often with the introduction of GitHub Copilot. We also engaged with DevEx and DevOps metrics vendors to understand both the challenges and successes they've experienced while helping customers elevate engineering performance or to justify the investment in GenerativeAI. So these steps were created to balance the inherent complexity of engineering success with practical, achievable steps for teams, including those earlier in their improvement journey.

In this playbook, we'll outline suggested metrics to monitor as part of your improvement efforts for each zone. Keep in mind that these metrics are downstream, or lagging metrics, and in the majority of cases should be complemented with leading metrics. Both leading and lagging metrics may be measured using telemetry and/or survey data, depending on your context, and the way these metrics are calculated will depend on your teams' engineering workflows and the systems supporting them–for example, you may use Jira or ServiceNow alongside GitHub.

As you dig into this playbook, we encourage you to keep a few concepts in mind:
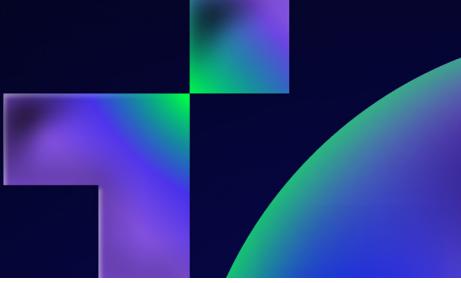
- Always bring a team perspective to improvement
- Select and use metrics with care to avoid gamification
- Balance the cost of measurement with the benefits of measurement
- Focus on improvements over time rather than overindexing on benchmarks

Engineering teams have the potential to fuel incredible change and accelerate business outcomes. With GitHub's ESSP, you can unlock engineering's potential through creating a culture of excellence that inspires and supports engineers to do their best work.

What are the GitHub engineering success zones?

GitHub's zones can be understood as a layered system: business outcomes sit at the top, supported by a foundation of quality, velocity, and developer happiness. Shaped by leading DevEx and DevOps metrics frameworks like SPACE and DevEx, DX Core 4, and DORA, together, they offer a practical and holistic view of your engineering system.

For each zone, GitHub suggests three downstream metrics that you can monitor to improve your team's engineering performance, as shown in the figure below. While these metrics are from industry best practices and are appropriate for many organizations, per SPACE, there can be reasons why an organization may prefer different downstream metrics.

GitHub's engineering system success metrics
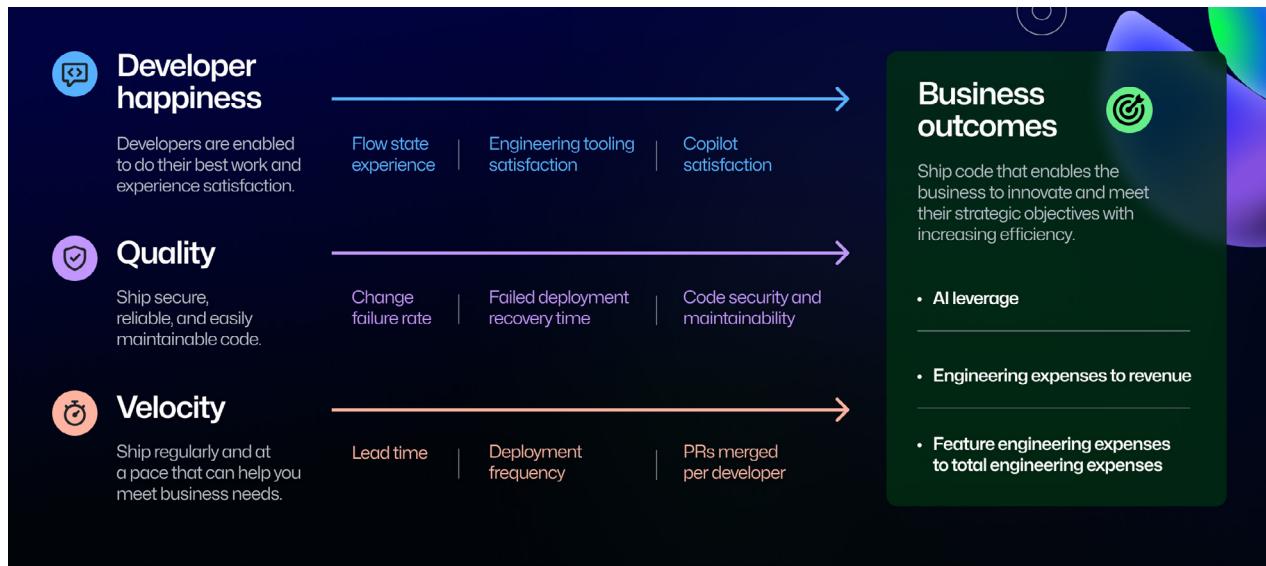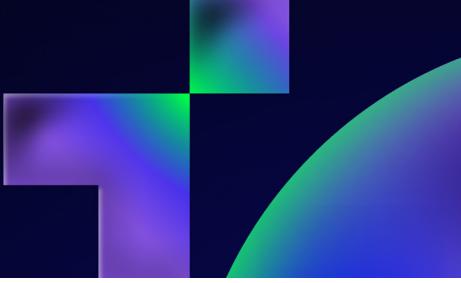
Fig 1: GitHub's engineering system success metrics

Sustainable improvement in any of these metrics will generally take months. We recommend using leading indicators–metrics that are likely to change faster–in addition to these downstream metrics. Need some help choosing your leading metrics? We'll give you more guidance below.

ESSP: Building on SPACE for Engineering Excellence

The SPACE framework provides a comprehensive approach to measuring and improving developer productivity. By capturing metrics across multiple dimensions, teams can develop a holistic view of their engineering effectiveness. The framework recommends measuring at least three of these key dimensions:

- **Satisfaction:** Measures developer satisfaction with tools, processes, and work environment.
- **Performance:** Evaluates the outcomes and quality of development processes, focusing on both individual and team-level achievements.
- **Activity:** Counts measurable development actions like pull requests, commits, and code reviews.
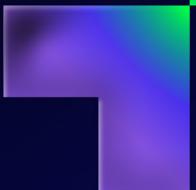
- **Communication and collaboration:** Assesses how effectively team members and code work together, including cross-functional interactions, knowledge sharing patterns, and API usage.
- **Efficiency and flow:** Tracks system throughput and developer time, measuring both process efficiency and developers' ability to maintain focus.

The ESSP builds upon SPACE by identifying 12 specific metrics that help teams improve engineering system performance. While the ESSP organizes these metrics along developer happiness, quality, velocity, and business outcomes, they map directly to the SPACE framework's holistic approach.

How to calculate your 12 metrics

Much of how to calculate the 12 metrics will depend on your engineering workflows and ecosystem. For example, your tech stack will influence how to measure each of these metrics. Perhaps you rely on tools beyond GitHub—like Jira or your incident management system—to calculate metrics like lead time or failed deployment recovery time. It's also important to understand your teams' workflows to determine which data to use from GitHub or other data sources in your engineering system. For example, what do you consider to be a production failure, and what data source in your engineering tools best reflects this definition? Similarly, what is your definition of "in production?"

Some metrics, like satisfaction with tooling, are ideally suited for developer surveys. Surveys can also be a practical choice for metrics like change failure rate—offering valuable insights without the need for telemetry. Developers are well-equipped to provide such information, and engineering leaders may decide that the benefit of calculating a metric through telemetry doesn't outweigh the cost and complexity. Organizations that don't yet have mature DevEx and/or DevOps metrics tooling may find surveying a particularly useful option as they start their transformation journey. DevEx and DevOps metrics vendors can assist with compilation of these metrics where an organization does not have this capability.

In using the ESSP metrics it is important to consider these two ideas:

- **There will be multiple factors (many outside GitHub) that will impact performance improvement.** Tooling, team processes and culture, and contributions to the software development lifecycle beyond engineering (i.e. prioritization processes, incident responses processes) may impact performance in some metrics.
- **We believe that engineering is a team sport.** User-level usage metrics can provide insights into how individual developers are engaging with tools, helping organizations to support engineers to make the most of available resources. However, it's crucial to approach user-level metrics with care, as misuse, unfair assumptions, or a one-size-fits-all mindset can result in overlooking the diversity of roles and contributions within a team. Depending on their specific job function, developers will face different challenges, and it's critical to account for these nuances. For engineering system metrics we recommend that you focus on teams and organizations, rather than scrutinizing individual developers. Using metrics to single out developers or enforce rigid standards can erode trust and undermine the collaborative culture essential for engineering success.

Leading versus lagging metrics

In the GitHub ESSP, balancing both leading and lagging indicators and using companion metrics is essential to achieving engineering system performance improvements.

- **Lagging indicators**—often synonyms with downstream metrics—reflect outcomes, such as deployment frequency and mean time to recovery, that are measured after work is completed. These lagging metrics are key to understanding long-term results, since gains often take time to be realized.
- **Leading indicators**—typically closer to the source of friction, provide early signals about areas that may impact downstream metrics later on. For example, improvements in code review time alongside developer confidence in the code review process can signal potential improvements in deployment speed or quality. To truly measure progress, it's important to complement each of the 12 engineering success metrics with leading indicators that reflect the team's day-to-day coding activities and the points of friction to be addressed, allowing for proactive adjustments. Depending on your friction points, the SPACE Communication and Collaboration domain is important to consider as part of leading indicators selection. This balanced approach helps teams anticipate issues, validate progress, and ensure continuous improvement in alignment with the playbook's goals.

- **Companion metrics**–Companion metrics are supplementary indicators that provide context to a primary metric, offering a more rounded understanding of performance. For instance, while lead time is a metric for assessing velocity, it can sometimes be misleading if used alone. Adding companion metrics like change failure rate helps to clarify if shorter lead times reflect actual improvements or if there's a trade-off, such as decreased quality due to rushed deployments. However, it's essential to strike a balance; too many companion metrics can dilute focus, and increase measurement costs, while too few can risk misinterpretation or misuse of the primary metric.

Metric recommendations based on zone

Quality

Change failure rate

- **SPACE dimension:** Performance
- **Definition:** The percentage of changes to production or released to users that result in degraded service [2]
- **Improvement (positive) direction:** Decrease is good
- **Link to business outcomes:** Lower change failure rate may mean higher reliability and fewer disruptions for customers
- **Calculation advice:** What events are considered a production deployment? What event signals service failure?

(Median) Failed deployment recovery time
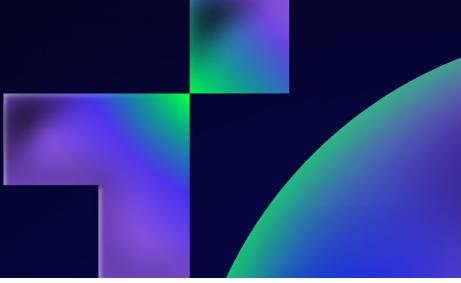
- **SPACE dimension:** Efficiency and flow
- **Definition:** How long it takes an organization to recover from a failure in production [3]
- **Improvement (positive) direction:** Decrease is good
- **Link to business outcomes:** Faster recovery from deployment failures may mean reduced downtime and maintains customer trust.
- **Calculation advice:** What event signals service failure? What event signals that the failure is resolved?

(Median) Code security and maintainability

- **SPACE dimension:** Performance
- **Definition:** Degree of threat resilience and minimized risk exposure, and ease of codebase maintenance, adaptability, and extension
- **Improvement (positive) direction:** Increase is good

2   From DORA: https://dora.dev/quickcheck/
3   From DORA: https://dora.dev/quickcheck/

- **Link to business outcomes:** Enhanced code security and maintainability may reduce risks, lower costs, and support ongoing innovation.
- **Calculation advice:** What event signals code vulnerability and exposure threat? What quality attribute signals code maintainability? What quality attribute signals code adaptability and reusability?
- **Need to know:** Availability of telemetry or survey data to evaluate both code maintainability and security
- **Tips:** This metric could be calculated through a combination of analytics from GitHub Advanced Security, SonarQube, or similar products or based on survey data. [4]

Velocity

(Median) Lead time

- **SPACE dimension:** Efficiency and flow
- **Definition:** The amount of time it takes a commit to get into production [5]
- **Improvement (positive) direction:** Decrease is good
- **Link to business outcomes:** Shorter lead times may enable faster responses to market demands.
- **Calculation advice:** What event signals first commit for a release? What event signals successful production deployment (consider incremental release handling)?

Deployment frequency

- **SPACE dimension:** Activity
- **Definition:** How often are releases deployed to production [6]
- **Improvement (positive) direction:** Increase is good
- **Link to business outcomes:** Higher deployment frequency may enable rapid innovation and faster customer feedback cycles.
- **Calculation advice:** What event signals successful production deployment (consider incremental release handling)

(Mean) PRs merged per developer

- **SPACE dimension:** Activity
- **Definition:** Number of pull requests successfully merged divided by total developers
- **Improvement (positive) direction:** Increase is good

---

4   Survey questions can support answering this question where telemetry is not available, for example: It's easy for me to understand and modify the code that I work with.

1 = Never; 2 = Rarely; 3 = Sometimes; 4 = Very Often; 5 = Always (This question is from DX's Developer Experience Index (used with permission))

5   From DORA: https://dora.dev/quickcheck/

6   From DORA: https://dora.dev/quickcheck/

- **Link to business outcomes:** Higher PR merge rates per developer may indicate effective collaboration and accelerated delivery.
- **Calculation advice:** How many developers to include in calculation? What event signals that the PR is merged?
- **Tips:** Focus on total PRs rather than calculating average for an individual and then calculating the mean. GitHub recommends taking particular care in the calculation of this metric. It should not be used to compare engineers to one another. Instead, the metric's purpose is to provide a measure of output adjusted for the number of engineers working within a team or organization.

Developer happiness

(Median) Flow state experience

- **SPACE dimension:** Efficiency and flow
- **Definition:** I have significant time for deep, focused work during my work days. 1 = Never; 2 = Rarely; 3 = Sometimes; 4 = Very Often; 5 = Always [7]
- **Improvement (positive) direction:** Increase is good
- **Link to business outcomes:** Improved flow state experience may enable engineers to deliver same or higher-quality work faster, with fewer errors and interruptions.
- **Calculation advice:** Ordered survey responses for organization or team. Identify middle value in results. Learn more about developer flow.

(Median) Engineering tooling satisfaction
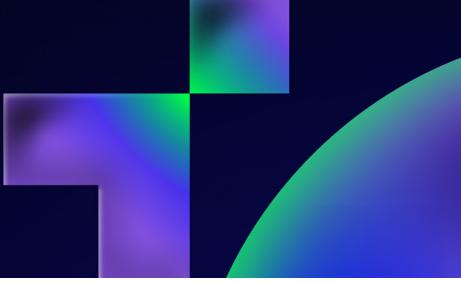
- **SPACE dimension:** Satisfaction and well-being
- **Definition:** How would you rate your overall satisfaction with the engineering tooling you use? 1 = Very unsatisfied, 2 = Unsatisfied, 3 = Neutral, 4 = Satisfied, 5 = Very satisfied [8]
- **Improvement (positive) direction:** Increase is good
- **Link to business outcomes:** Greater satisfaction with engineering tooling may reduce friction, enabling faster and higher-quality software delivery.
- **Calculation advice:** Ordered survey responses for organization or team. Identify middle value in results.

(Median) Copilot satisfaction

- **SPACE dimension:** Satisfaction and well-being
- **Definition:** If you have been assigned a Copilot license, how would you rate your overall

---

7   This question is from DX's Developer Experience Index (used with permission)
8   This question is from DX's Developer Experience Index (used with permission)

satisfaction with Copilot? 1 = Very unsatisfied, 2 = Unsatisfied, 3 = Neutral, 4 = Satisfied, 5 = Very satisfied, NA

- **Improvement (positive) direction**: Increase is good
- **Link to business outcomes**: Higher satisfaction with Copilot may be linked to improved velocity or quality outcomes.
- **Calculation advice:** Ordered survey responses for organization or team. Identify middle value in results
- **Tips:** This question should only be made available to staff with a Copilot license, or results from non-Copilot license holders omitted from the calculation.
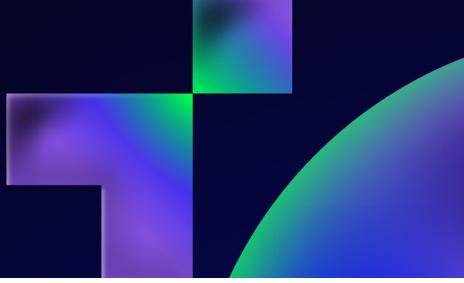
Business outcome

(Percentage) AI leverage

- **SPACE dimension**: Activity
- **Definition**: Opportunity being realized due to effective engagement with AI, through calculating the difference between potential and current AI-driven productivity gains across employees working in engineering.
- **Improvement (positive) direction**: Increase is good
- **Link to business outcomes**: Higher AI leverage may reduce manual engineering effort, or accelerate or enhance the quality of delivery with increased cost efficiency.
- **Calculation advice:** Average time-savings associated with AI use. Average staff salary per week. Total staff who could benefit from AI in engineering. Total staff currently 'engaged' with AI for engineering. Cost of AI per week
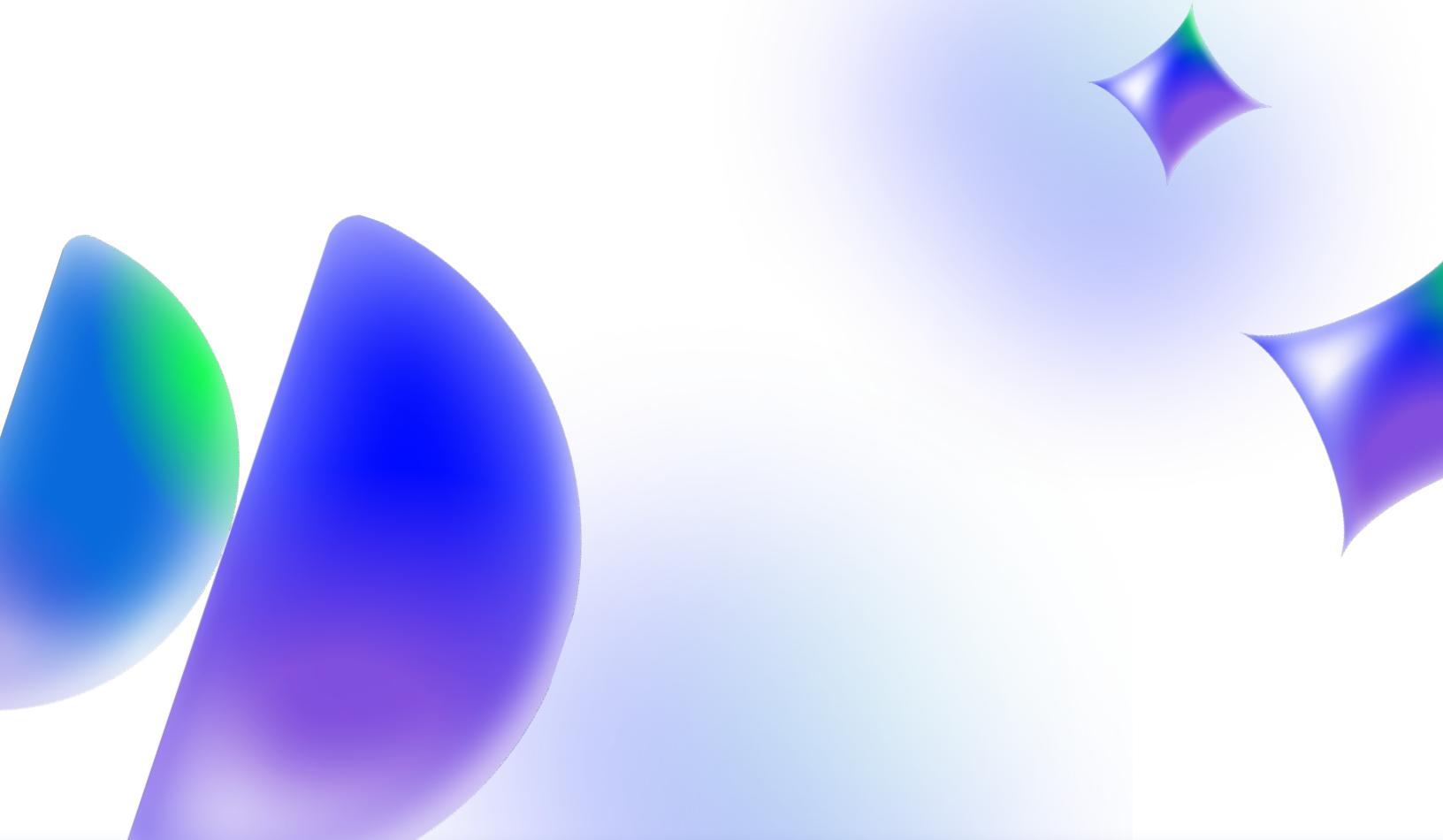
(Percentage) Engineering expenses to revenue
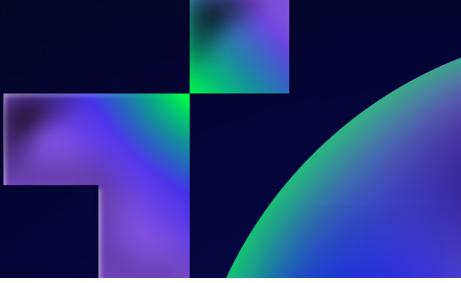
- **SPACE dimension**: Performance
- **Definition**: The total engineering spending as a proportion of an organization's total revenue.
- **Improvement (positive) direction:** Decrease is good
- **Link to business outcomes**: Lower engineering expense ratios may indicate efficient engineering investment and increased profitability.
- **Calculation advice:** What expenses are considered 'total engineering'? What constitutes organizational revenue?
- **Tip:** Best monitored at organizational-level rather than team-level.

(Percentage) Feature engineering expenses to total engineering expenses

- **SPACE dimension:** Performance
- **Definition:** The proportion of engineering expenses for feature development as a portion of total engineering expenses.
- **Improvement (positive) direction:** Increase is good
- **Link to business outcomes:** Higher allocation to feature engineering expenses may allow more direct investment in customer-facing improvements that drive revenue growth.
- **Calculation advice:** What expenses are considered 'feature development'? What expenses are considered 'total engineering'?
- **Tip:** Best monitored at organizational-level rather than team-level

Three steps to engineering success
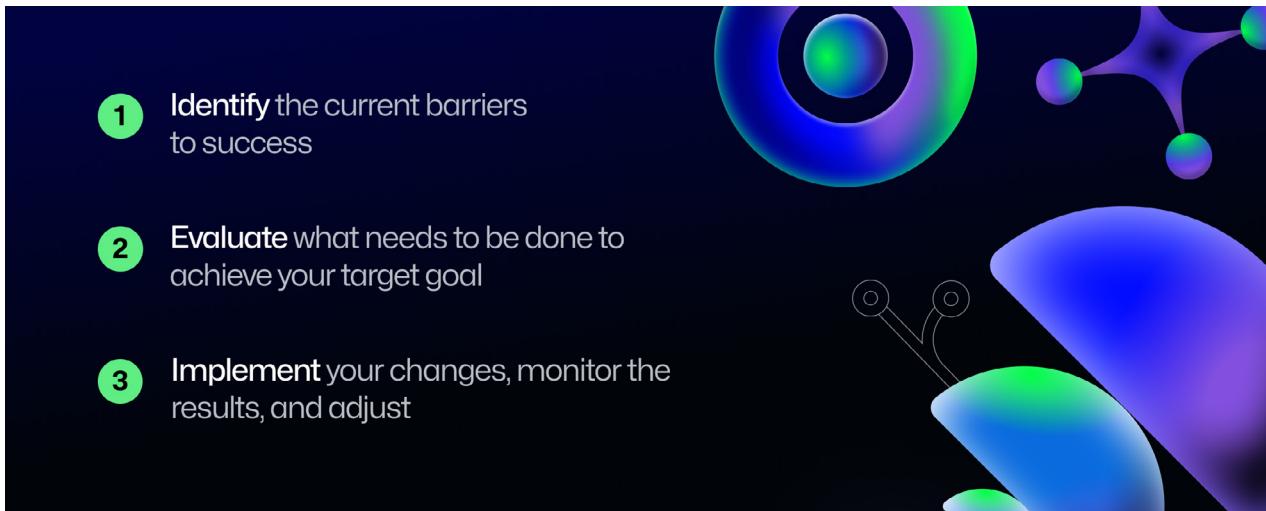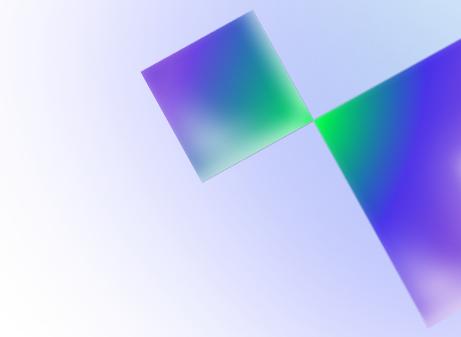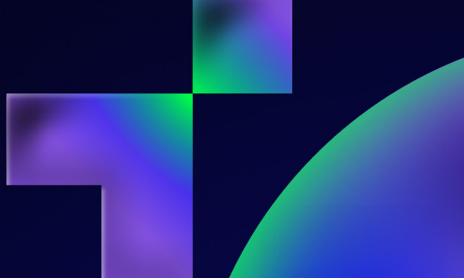
Fig 2: Three steps to engineering success

These three steps are the heart of GitHub's ESSP, as they highlight your current friction points and manage your expectations for how changes will drive improvements. As you consider your future state, GitHub recommends thinking across the zones: quality, velocity, developer happiness, and how together they contribute to business outcomes.

As part of the three-step process, GitHub also recommends the use of leading indicators—like close to code telemetry such as number of commits, and surveys—to monitor the early impact of the agreed changes on your engineering system. Your choice of leading indicators will depend on the friction points being addressed.

Step 1: Identify the current barriers to success
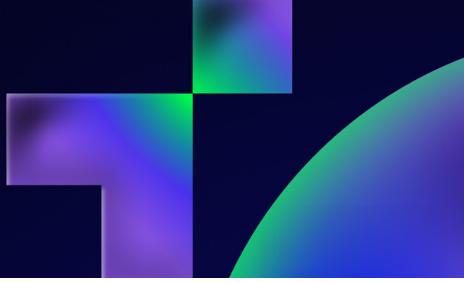
The purpose of step one:

The goal of this step is to develop a clear understanding of the obstacles preventing improvements. By understanding your current state and your desired future state, and the gaps and barriers to reach the future state, teams can prioritize areas that need attention and ensure changes are targeted and effective. This step encourages understanding of your current performance baseline. That being said, if these current performance baselines have not yet been quantified, it is still possible to start working towards improvements.

Tasks for step one:

1.1 Audit current processes, gather data, and understand organizational priorities:

- **Build an understanding of your development lifecycle:** Put together a complete picture of your teams' SDLC processes and workflows, from idea to ship to learn. Identify the different tasks and process flows, while also recognising that teams may have different development lifecycles. Understanding the lifecycle is an essential requisite to calculating metrics and determining bottlenecks. Need help charting your lifecycle? There are many different ways to chart your lifecycle. Check out GitHub's documentation on building diagrams.

- **Gather available metrics:** Collect your team's data on existing metrics for the zones, so that you have a baseline. You don't need advanced telemetry data to get started: qualitative insights from developer surveys or focus groups can offer initial baselines. These qualitative baselines capture team sentiments and highlight areas needing attention. As you progress, you can make a plan to incorporate quantitative data to refine your baselines and expand your view. By regularly reviewing progress against your baselines, your organization can make informed decisions, adjust strategies proactively, and celebrate tangible achievements on your path to engineering success.

- **Industry benchmarks:** Benchmarks are reference points drawn from industry data, often representing average performance, or higher percentiles such as P75 or P90, for specific metrics (See the DX Core 4 benchmarks and the DORA report benchmarks). While benchmarks can reveal how your team's performance compares to others, remember to take into account differences in team workflows. There is benefit in focusing on improvements over time rather than benchmarks.

- **Understand zone priorities:** Engage with stakeholders to clarify which zones are currently most critical for the organization given your business goals. Remember that developer happiness is just as critical as the other zones. This helps to align the team's efforts with business goals and strategy.

1.2 Conduct qualitative research:
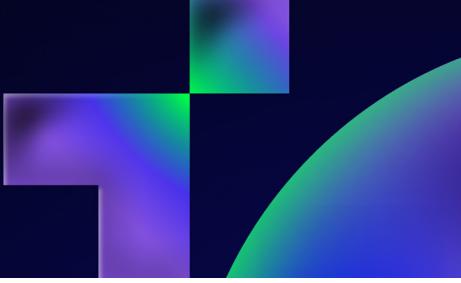
- **Gather feedback:** Interview or survey developers and other key stakeholders to understand their pain points across the development lifecycle, remembering to include the outer loop.
  - Focus on where friction exists: where there seems to be delays and what impacts engineers' satisfaction. For organizations with more mature analytics capabilities, you may be able to go beyond the recommended zone metrics to understand more granular trends associated with your development lifecycles, like periods of delay when progressing a pull request from submission to merge.
  - Make sure that you're seeking information on cultural, social, or process factors that may affect the development lifecycle. Are team members feeling supported and motivated? Is there a mindset that's adversely impacting quality or velocity? Are internal tools or processes slowing down work?

1.3 Prioritize key metrics and barriers:

- **Map findings to the zones:** Categorize each identified barrier by which zones it impacts and onto your developer lifecycles.
- **Prioritize the metrics to target:** Once barriers are identified, prioritize which metrics should be targeted for improvement. Consider any trade-offs between elements of your desired future state and the barriers that are most actionable, keeping in mind your business goals.

Tools needed for step one:

- Analytics and metrics dashboards
- Survey and feedback tools (to support focus groups, interviews, etc.)
- Process mapping tools
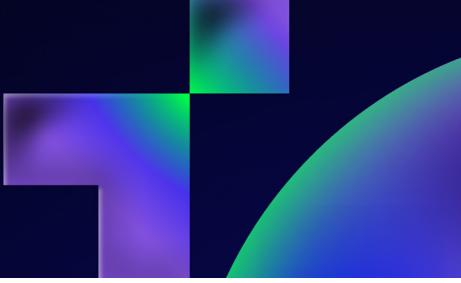
Skills needed for step one:

- Data analysis skills
- Stakeholder engagement
- Technical acumen and root-cause analysis

Tips for a successful step one:

- **Focus on root causes, not just symptoms:** While undertaking research, avoid being misled by surface-level issues. For example, slow velocity might be attributed to manual testing, but the root cause could be a lack of trust in automated testing. Dig deeper to uncover the underlying problems. Common antipatterns in software engineering can be a great place to start.
  - A note: Antipatterns are common solutions to common problems where the solution doesn't actually resolve the problem and may inadvertently cause undesired consequences. Check out this GitHub resource on antipatterns for a detailed look into how they might manifest within your team.
- **Involve the right people:** During tasks 1.1 and 1.2, gather input from various roles such as developers, testers, operations, security, and product managers to ensure a comprehensive view of the workflow. This prevents overlooking critical perspectives or bottlenecks.
- **Balance quantitative and qualitative data:** Metrics alone don't tell the full story. Make sure data-driven analysis includes feedback from the team to capture cultural and morale-related barriers that may not appear in the numbers. [Learn more](#) about the value of both qualitative and quantitative data to improve engineering system success.
- **Don't overwhelm yourself with too many barriers:** Focus on the most impactful barriers rather than trying to tackle everything at once. Prioritize key areas that will provide the greatest momentum towards your future state.
- **Ensure psychological safety:** Create an environment where team members feel safe enough to share their frustrations and challenges without fearing repercussions. This fosters honesty and leads to better insights on the true barriers.
- **Compare for learning, not judgement:** While it can be valuable to compare trends in teams' metrics and workflows, keep in mind that teams may have different contexts, work styles, and challenges. Use comparisons to identify best practices and areas for improvement, rather than as a direct performance measure. Encourage knowledge-sharing on what's working well, but be mindful that what works for one team may not always apply to another due to differing goals, technologies, or constraints. This is where qualitative information can be particularly useful.

# How GitHub understands and prioritizes opportunities for improvement
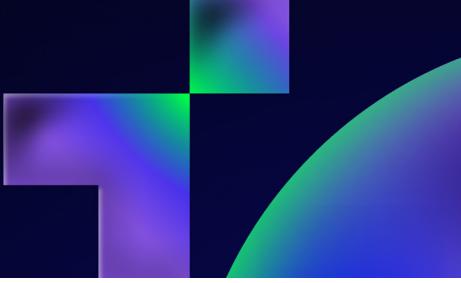
Quality is a very important zone for GitHub. The importance of this zone is evident in conversations across the organization, including our leadership team. But anecdotally, we felt that our change failure rate and time to restore service could be improved. Our first step was to gather baseline data to measure both metrics. We gathered data from our internal incident management tooling to understand the number of incidents that were declared, along with the time between an incident beginning, the time it was declared, and when the incident was resolved. We also gathered metrics from our defined service-level objectives (SLO) to understand which SLOs represented change failure rate, and measured which services were more frequently impacted.

As part of considering our current performance relating to quality, GitHub identified where the potential bottlenecks or friction fall in our development processes. First, we identified that there were some scenarios where deployed code changes would create an incident, and reverting the changes took longer than we would like and ultimately increased our time to restore service metric. Second, we also analyzed data from our internal developer satisfaction survey — which asks engineers questions about their satisfaction with incident response tooling, testing, and validation capabilities — and their confidence in being able to respond to incidents.

The insights from these surveys revealed time delays in rolling back deployments, which introduced failures. We increased our understanding of these developer reports by triangulating their feedback with quantitative data.

We also recognize that as we continually improve quality, we want to maintain velocity. Our developer satisfaction survey showed that although our deployment frequency metric was well in line with our organizational targets, (GitHub typically deploys approximately once per hour), our developers were dissatisfied with the experience of deploying their code.

For example, being on standby for an unknown amount of time waiting for deployment to start impacted their flow. This dissatisfaction, coupled with our median lead time metric suggested that changes here could increase our overall velocity, and potentially increase developer happiness. While GitHub identified room for improvement on both the quality and velocity zone, we ultimately prioritized improvements to quality over velocity, as it was most critical to achieving our business objectives.

Step 2: Evaluate what needs to be done to achieve your target goal

Purpose of step two:

The goal of this step is to identify, evaluate, and agree on changes that could address the barriers identified in step one. By doing this, teams can determine the most effective way to achieve their future state and drive improvements in business outcomes. The focus is on identifying actionable changes that are aligned with team goals and organizational priorities, ensuring interventions lead to tangible, sustainable improvements. These changes may be technology changes or additions, but they may be cultural, social, or process-related changes, too.
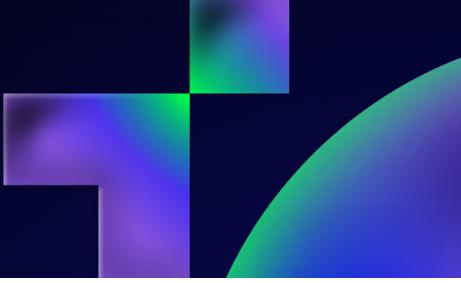
Tasks for step two:

2.1 Evaluate and prioritize changes:

- **Identify potential solutions:** Based on the barriers identified in step one, begin by brainstorming possible changes that would reduce each barrier. For example, if a barrier relates to slow deployments due to manual processes, one intervention might be implementing automated deployment pipelines. If developer happiness is low, consider initiatives that address workload balance or provide better tooling.
- **Estimate cost and/or resource requirements:** For each intervention, estimate the resources required, including time, personnel, tooling, and budget. Consider both the initial implementation effort and ongoing effort. Use this to support evaluation of the feasibility of each intervention.

2.2 Conduct a risk, cost, benefit analysis for the changes:

- **Identify risks:** Each change will have risks. For instance, automating a process may inadvertently introduce new errors or bugs if not tested thoroughly. For cultural changes, risks might include pushback from the team or slow adoption. Assess the potential risks for each change, including both technical risks and people-related risks.
- **Weigh the benefits against the risks and costs:** For each change, clearly outline the expected benefits and how they will support achievement of the future state. Make sure to balance this with any potential negative impact on other areas of the business (e.g., increasing velocity at the cost of quality or developer happiness). Also account for the

cost and/or resource implications identified in task 2.1.
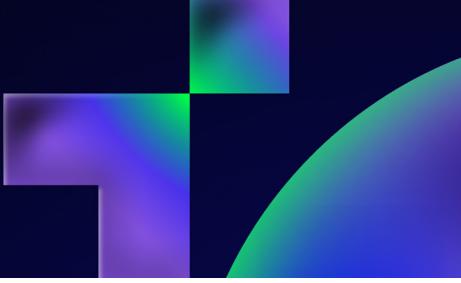
- **Start with a pilot:** For significant changes or changes with high risk, consider starting with a pilot. Test the solution with a few teams or using a smaller subset of the process before scaling across the organization. This allows for faster learning and iteration, and reduces the chance of large-scale disruption.
- **Create a mitigation plan:** For high-priority changes with notable risks, develop a risk mitigation plan. This could involve rolling out the intervention in phases or involving additional stakeholders to ensure the solution is robust.

2.3 Engage with key stakeholders:

- **Review with teams:** Share the proposed changes with engineering teams to get feedback. Are the changes realistic? Will the changes support long-term goals, or are there concerns about their implementation? Developers, testers, product managers, and other team members will have unique insights into the practicalities of implementing changes. If you have undertaken a pilot, share the findings from the pilot.
- **Secure buy-in:** For more significant changes, secure buy-in from leadership and other stakeholders. Present the expected benefits alongside the potential risks and cost or resource requirements. It's important that there is alignment across all levels of the organization, especially when the interventions involve process changes or resource investments. Also be realistic about the timeframe for implementation and the realization of benefits.
- **Incorporate feedback:** Be open to adjusting interventions based on feedback from stakeholders, including those involved in any pilots. Some changes may need to be deprioritized if they are deemed too risky or resource-intensive, while others may be refined based on team input.

Data needed for step two:

- Barriers and priorities from step one
- Information on potential changes
- Information on available resources, budgets, etc.
- Outcomes from any pilots
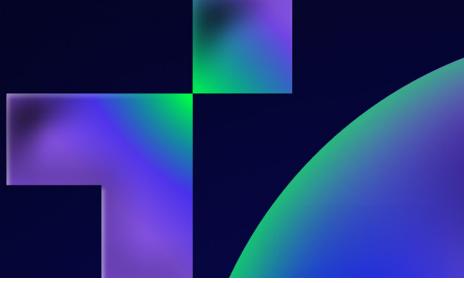
Skills needed for step two:

- Business case development (cost, risk, and benefit analysis)
- Stakeholder engagement
- Technical acumen and root-cause analysis
- Research skills to explore potential change options
- Coaching skills to steer and co-create desired behaviors in the pilot

Tips for a successful step two:

- **Don't forget long-term sustainability:** Even though it can be tempting to focus on quick wins, make sure the selected changes are sustainable long-term. Avoid changes that solve short-term problems but create additional maintenance burdens down the road. For example, deploying new tools or software across the organization may immediately accelerate velocity, but without investing in training, support, and change management strategies, it can lead to frustration, errors, and reduced performance.
- **Consider trade-offs across zones:** Remember that changes may affect more than one zone at once. Make sure that changes to improve one zone (such as velocity) do not significantly negatively impact another (such as developer happiness or quality).
- **Involve your team early:** Changes are more likely to succeed if they're co-created with the team. Avoid imposing top-down changes without gathering input from those who will be most impacted.
- **Identify success metrics:** Before implementing any changes, define how success will be measured. Establish which metrics or indicators will show that the intervention is leading towards your future state. Consider both leading and lagging indicators for your target future. For example, a reduction in deployment time may be your lagging indicator, but developer perception of PR duration and reduction in PR dwell time are leading indicators.
- **Stay agile and iterative:** Don't wait until you have the perfect solution to implement changes. Adopt an iterative approach where small changes can be tested with leading indicators, refined, and scaled over time. This reduces risk and ensures that the team can pivot if an intervention isn't yielding the expected results.
- **Focus on high-impact, low-effort wins:** If your team is overwhelmed by potential changes, start with the solutions that are both easy to implement and have high potential impact. These can provide immediate wins and build momentum for tackling larger, more complex barriers.

How GitHub identifies and pilots changes

With a baseline established and hypothesis on the key bottlenecks relating to quality improvements, we assigned a team to dig deep on deployment rollbacks. The team proposed a few changes that would allow for earlier detection of production issues, and opportunities to respond to issues more quickly. Each proposal was estimated based on level of effort and level of impact.

These changes included extending a wait time between deployments to allow for more time to test code in between deployments, ultimately making a code rollback easier. This was a very low effort change, with a potentially high impact. The team also proposed making changes to how rollbacks were triggered and executed, which reduced the amount of time a rollback would take, thus improving time to restore. This was a medium effort change, but the potential positive impact was deemed high.
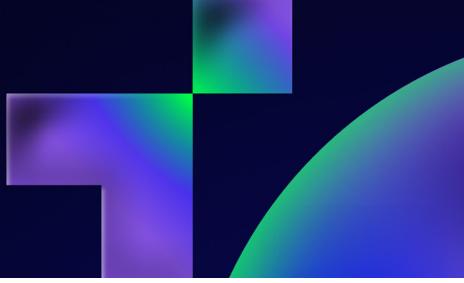
The team also proposed strategies to detect change failure earlier in the process, including:

- Implementing an end-to-end testing strategy during deployments (leveraging GitHub Actions),

- A stage-based deployment model, which would deploy code to internal staff before deploying to customers

- An automated error detection system, which would alert when new exceptions were detected during a deployment.

The team also recognised the value of our secret scanning and code scanning features, and sought to embed them even more deeply in our practices. These suggestions were made by consulting with many teams and experts, including application developers, observability teams, reliability teams, and delivery teams.

In parallel, and based on feedback from developers about difficulties in responding to unpredictable and confusing deployments, the team proposed simplifying notifications, surfacing helpful log messages during the deployment process, and streamlining the UI. We also saw an opportunity to improve the developer experience by increasing our transparency for when a deployment was likely to start, and to enhance the monitoring experience during the deployment.

It was important to weigh these proposals against the potential risks. Some of the proposed plans required slowing down deployments, which would increase the mean lead time for changes, and

reduce deployment frequency metrics in our velocity zone. We weighed the impact to our velocity and determined how much of an impact we were able to withstand in order to see gains in quality metrics. To counteract some of the reductions in velocity, the team also proposed increasing the number of changes that could be deployed at once. This analysis was done very carefully to make sure that an increase in changes would not result in a reduction of quality.

Once we decided which process and tooling changes to pursue—just like our features—we then took an incremental approach to roll-out. You can work in incremental changes in two different ways:

- The number of changes you make at a given time. Keep the scope small so you know which change is driving what impact
- In terms of the distribution of the change (then scale to build confidence)

We also used a test application that allowed us to A/B test our process and tooling changes to more accurately understand their impact on key metrics.

Step 3: Implement your changes, monitor the results, and adjust
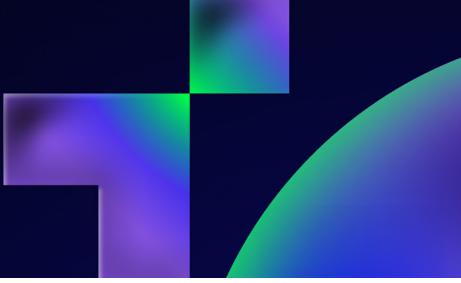
Purpose of step three:

The goal of this step is to scale the prioritized changes, including monitoring the progress towards reaching your target future state. Successful implementation requires ongoing monitoring and willingness to adjust to make sure changes are delivering the desired improvements and are contributing to your business outcomes. By tracking performance and iterating as needed, teams can make sustained progress and avoid regressing.

Tasks for step three:

3.1 Implement the changes:

- **Assign ownership and responsibilities:** Ownership ensures accountability and makes it easier to monitor progress, so each intervention should have a clear owner responsible for its implementation and success. The owner may be a developer,
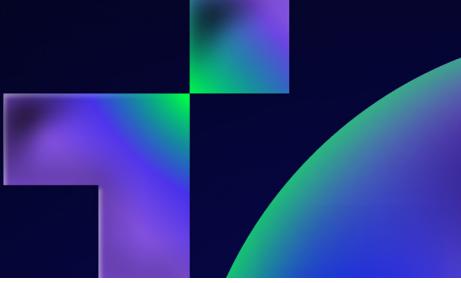
engineering lead, or cross-functional team, depending on the nature of the intervention. Link the implementation of changes to senior or executive KPIs (key performance indicators) or targets.

- **Foster communication and transparency:** Clearly communicate to all relevant stakeholders when an intervention is being rolled out, why it's being done, and what the expected outcomes are. This transparency fosters trust and encourages team members to actively support the change. Encourage feedback during the rollout to help identify any immediate issues or resistance. Remember that tooling or technology changes also often require accompanying policy, process, or cultural changes.

- **Train teams when necessary:** Some changes may require new skills or processes. For example, if implementing a new automated deployment pipeline, make sure teams are trained in how to use the tooling. Offer support and guidance to reduce friction during adoption.

3.2 Monitor performance post-implementation:

- **Track key metrics:** Once changes are implemented, track the identified metrics across the zones. Compare the new metrics with the baseline established in step one to evaluate the impact of the intervention. However, be realistic about the time it takes for metrics to shift and expect some variance in performance rather than consistent gains. Most change initiatives will require the use of a set of leading indicators. Often, qualitative data like surveys are a useful leading indicator in addition to close to code metrics such as pull request review times, depending on the current and future state and barriers being addressed by the changes. Learnings from any pilots can be useful in understanding likely timeframes to achieve downstream improvements.

- **Gather qualitative feedback:** In addition to metrics, gather feedback from developers, operations, and other stakeholders on how the changes are impacting their day-to-day work. Use interviews and team retrospectives to understand whether the changes are positively affecting team morale, collaboration, or overall satisfaction.

- **Identify early wins and challenges:** Keep an eye out for both early successes and challenges. Celebrate small wins, such as reductions in pull request review times or improved test coverage, to build momentum. On the flip side, be prepared to identify and address any resistance or unforeseen issues early, before they grow into larger problems.

3.3 Adjust and iterate:

- **Analyze what's working and what's not:** After an initial period of implementation, review the data and feedback to confirm that the changes are having the desired effect. Are the target metrics improving? Are there trade-offs that need to be reconsidered? For example, are quality targets being maintained while velocity is improving? It's essential to critically assess whether the changes are solving the barriers identified in step one.
- **Pivot if necessary:** If changes are not delivering the expected results, don't hesitate to pivot. It's better to adjust mid-course than to persist with solutions that aren't working. Revisit the other potential actions from step two, and consider alternative approaches or adjustments.
- **Maintain continuous feedback loops:** Make monitoring and feedback an ongoing process. Don't treat implementation as a one-time effort. Use team retrospectives, stakeholder reviews, and performance dashboards to maintain a cycle of continuous improvement. Regularly check in on the health of the zones and be proactive in adjusting the changes as needed. Consider using automated alerting to make sure that if a metric is falling outside expected performance ranges, it can be reviewed and acted upon.

Tools needed for step three:
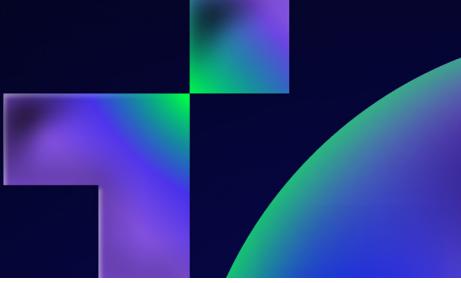
- Analytics and metrics dashboards
- Survey and feedback tools
- Project and change management tools

Skills needed for step three:

- Implementation management
- Data analysis and monitoring
- Change management
- Technical problem-solving and iteration

Tips for a successful step three:

- **Don't expect immediate perfection:** Not all changes will produce immediate or dramatic improvements. Be patient, and allow time for the changes to make a positive

impact. Surveys are a great tool for the earlier stages of an intervention. Remember, it may take time for the team to adjust and for the changes to be fully embedded.

- **Keep iterating on the changes:** Remember that even after successful implementation, further improvements can always be made. Teams should be encouraged to treat the process as ongoing and remain open to refining changes as new challenges arise. Changes in operating circumstances can also prompt the need to consider further iterations.

- **Watch out for unintended consequences:** Some changes may introduce new friction points or affect other areas of the workflow in unexpected ways. For example, speeding up deployments may lead to more frequent post-release bugs if the quality zone isn't balanced. Be vigilant in identifying these side effects and address them promptly.

- **Check in on psychological safety:** Make sure that teams still feel comfortable speaking up about issues post-implementation. Teams should feel empowered to offer honest feedback about what's working and what isn't, without fear of judgment.

- **Evaluate long-term impact:** Over time, make sure that the improvements are sustained and that new challenges aren't introduced. Look for enduring improvements in team performance and morale.

- **Use feedback for further learning:** [Treat failures as opportunities for learning](). If a change doesn't work, use the data and feedback gathered to understand why, and apply those

How GitHub implements change
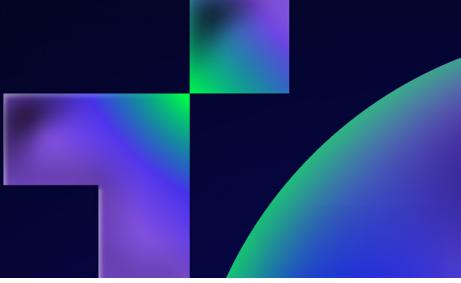
Following our incremental/piloting approach described above, GitHub very carefully rolled out the changes identified in step two to an increasing number of applications. Even beyond our piloting efforts, we still feature flag our process and tooling changes so that if an unexpected situation arises as we scale the rollout, we can quickly revert to the previous process/tooling.

For example, when we rolled out changes to the deployment pipeline, we changed how we measured deployment rollbacks. Previously, our metrics looked at a raw count of rollbacks, but given our intervention to deploy changes to staff before customers, we began tracking rollback metrics with a more granular view, measuring when a rollback included customer impact versus impacting internal staff. We also began tracking how soon issues were identified after a problematic deployment. This allowed us to show that the changes to the pipeline did improve our quality metrics, by completely preventing external incidents in some cases, and being able to respond to defects faster, thus reducing our change failure rate. Similarly, when implementing our end-to-end testing strategy, we were able to measure when the tests uncovered an issue that would have otherwise made it to production. This also reduced our change failure rate.

We also rolled out UI changes of our deployment tooling incrementally, which allowed the team to gather feedback and pivot approaches along the way. As part of the rollout, the team identified that while the UI improvements were helpful, some developers craved a more direct support model. In response, the team built alerts to proactively alert a support team if intervention is needed. While the UI could guide developers, the support model allowed for quicker resolution for more complex scenarios.

Maximizing the impact of your engineering system requires intentionality—which can be achieved through a systematic approach, a learning mindset, and investment in the tools, skills, and time needed to drive sustainable improvements.
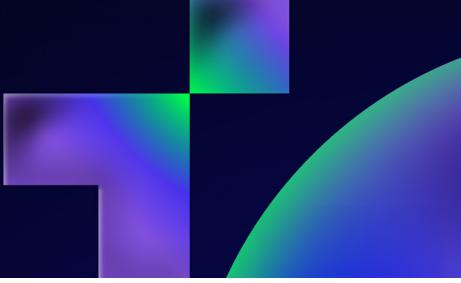
GitHub provides a suite of tools, including GitHub Copilot, to support you in achieving success in your software development, but these tools also need to be deployed intentionally regarding the problems you'd like to solve, and with an awareness that tooling changes often require social, process, and cultural changes.

A note on Copilot metrics:

The ESSP is part of GitHub's commitment to support our customers' understanding and growth of Copilot impact. GitHub will continue to connect with our customers to understand their highest data priorities, and develop and deliver a roadmap accordingly. Our current priority is to focus on exposing leading indicators of Copilot success (such as those on the Metrics API), which can be used alongside customer-sourced (or partner supported) lagging indicators.

When implementing GitHub Copilot, we recommend using leading indicators to guide your pilot and scaling efforts. Surveying developers on their experience with GitHub Copilot provides early insights into areas needing additional training, where GitHub Copilot is most beneficial, and potential time savings in achieving your engineering goals.

Beyond the steps: Make the playbook work for you

To drive meaningful and lasting improvements across the ESSP zones, GitHub recommends the three-step process. Yet, these steps alone aren't enough. As described in SPACE, achieving true engineering success also requires complementary principles, mindsets, and approaches that shape how teams interpret, implement, and sustain progress over time. Tailoring the playbook to your needs, fostering a growth mindset, and employing change management practices are essential to unlocking the full potential of the playbook in a way that resonates with your team — and that helps you achieve your unique goals.
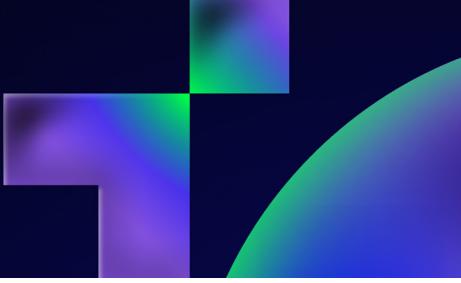
Below, we'll dig into additional concepts that will help support a well-rounded and adaptable approach to engineering success, and enable organizations to create a resilient, effective, and sustainable engineering system.

Tailoring

The GitHub ESSP should be tailored to align with your team's specific needs, workflows, and tooling. Rather than applying a universal approach, tailoring enables teams to select metrics that directly reflect their goals, context, and budget. For instance, some teams may choose to focus more heavily on developer happiness if morale is hindering engineering system performance, while others might prioritize velocity to meet business goals.

Tailoring also involves deciding how to measure metrics—opting for telemetry data when automated tracking is practical, or developer surveys when more nuanced feedback is needed. Measurement is a tool—valuable when it supports improvement efforts but not an end in itself. It's essential to invest only as much as needed to facilitate or evidence meaningful progress, avoiding the temptation to over-engineer the engineering measurement system.

Additionally, tailoring includes complementing downstream metrics (e.g., deployment frequency) with leading indicators that offer early signals of friction or improvement. By tailoring such elements, the ESSP can reflect the diverse ways that engineering teams work. As you tailor and select metrics, remember to collect data across the ESSP zones

(developer happiness, quality, velocity, and business outcomes) and include at least three of the SPACE dimensions (satisfaction, performance, activity, communication and collaboration, and efficiency and flow.)

Change management

Change management is essential for achieving success within the GitHub ESSP, ensuring that teams can adopt and sustainably use new metrics, tools, and practices effectively. Frameworks like ADKAR (Awareness, Desire, Knowledge, Ability, Reinforcement), Kotter's eight-step change model, or the three-stage change model associated with Lewin, provide structured approaches that can be tailored to guide engineering teams through change. For example, ADKAR's focus on awareness and desire is useful when introducing new metrics like those in the playbook, helping teams understand the role of these metrics in guiding sustainable improvement. Kotter's emphasis on building a coalition can rally support across teams, especially when adopting telemetry or new measurement methods. By applying these frameworks, change management can help teams feel prepared and supported.
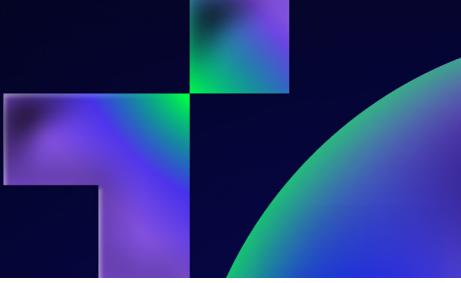
Growth mindset

GitHub's ESSP is most powerful when approached with a growth mindset that values learning as success in itself. This means recognizing that not every intervention will work as intended on the first try–and that's okay. Each attempt, whether it leads to immediate gains or requires recalibration, offers valuable insights that propel teams forward. By embracing the idea that failure is part of the process, teams can take bolder steps in understanding bottlenecks, experimenting with solutions, and refining their practices. This mindset fosters resilience, allowing teams to adapt, learn, and ultimately build an engineering culture where each iteration brings them closer to sustainable improvement.

Gamification

Gamification done right
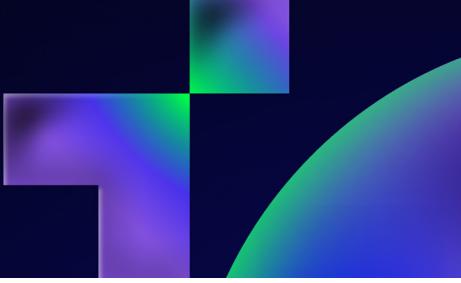
When thoughtfully designed, gamification can foster a positive and motivated engineering culture. Drawing from behavioral economics, gamification is more effective when it aligns with intrinsic motivators–such as the satisfaction of mastering a skill, solving complex problems, or contributing to team goals. For instance, rewarding developers with

recognition for maintaining high-quality code or celebrating team milestones fosters a sense of pride and commitment to excellence. Shout-outs to a team during a Town Hall or ensuring that efforts on strategic priorities are noticed in performance reviews can be motivating. When gamified elements focus on achievements that genuinely support developer happiness, quality, and velocity, they can enhance engagement and drive sustainable progress.

The pitfalls of gamification

However, it's also necessary to be aware of the risks of incentive misalignment, where rewards can encourage undesirable behaviors. For instance, using leaderboards to drive rapid code reviews may prompt rushed reviews that compromise code quality. Similarly, tracking and rewarding individual code contributions can lead to a focus on quantity over quality, potentially increasing technical debt. It's essential to recognize both subtle and overt incentives that the monitoring of metrics can introduce.

Alternatives to the GitHub engineering system success playbook

GitHub understands that organizational needs and situations vary. For customers who are looking for an alternative to the GitHub ESSP, here are a few strategies:

Understand feature usage without a focus on downstream impact

For organizations who may not have the capacity, or a pressing need, to invest in understanding the downstream impacts of engineering changes, we recommend focusing on feature usage telemetry and developer feedback. This option centers on gathering insights through developer surveys, focus groups, interviews, and usage telemetry, which will give you a closer-to-code picture of the developer experience. By focusing on these developer reports and usage-based data points, teams can still uncover actionable insights on developer satisfaction, and identify areas for improvement with a lower-level of analytics investment.

Business value engineering with a focus on delay, cost, and risk reduction

Business value engineering is useful for customers wanting to frame value and measure improvements across the dimensions of delay, cost, and risk reduction. It draws on developer-reported time savings (i.e. delay reduction) and other causally related metrics to provide upstream and downstream insights into GitHub Copilot adoption, usage, and the linking of downstream improvements specifically to GitHub Copilot.

Using SPACE as a foundation for your own framework

For organizations looking to develop their own approach to improving developer experience, the SPACE model provides a research-backed foundation for designing a holistic and balanced view of engineering. By structuring insights around the five

dimensions of SPACE (see Page 5), organizations can ensure they account for the complexity of engineering work at the individual, team, and system levels.

This approach helps avoid overly simplistic productivity measures and instead fosters a nuanced understanding of the factors that drive sustainable performance, collaboration, and satisfaction. By leveraging SPACE, organizations can shape initiatives that align with both developer well-being and business impact, ensuring an evidence-based path to engineering success.
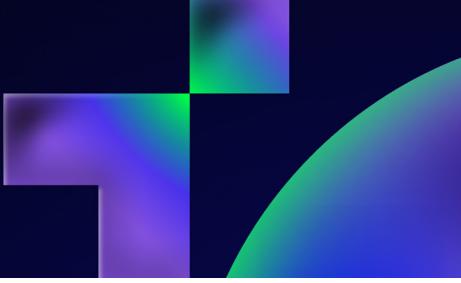
Stepping into action and towards success

Technology is always changing, but the steps outlined in GitHub's Engineering System Success Playbook (ESSP) are foundational and timeless. Whether you're keen to explore the potential of GitHub Copilot or simply need to unblock your team's workflows, the ESSP steps will guide you in identifying obstacles, implementing solutions, and continuously improving. The ESSP has an ethos of listen, act, learn—it's a pathway to unlocking your engineering team's full potential and driving remarkable business outcomes. Ready to elevate your engineering game? Dive into the playbook, start with step 1 today.

Want to learn more?
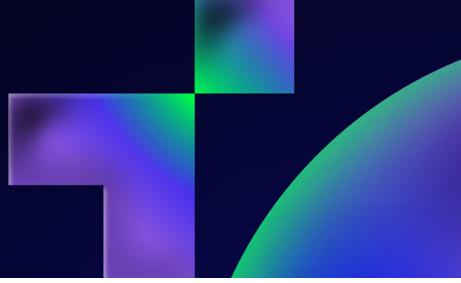
- GitHub Copilot onboarding survey:
  https://downloads.ctfassets.net/wfutmusr1t3h/6BD0BWsrVXIIq1gSnnsrUd/be55fd315df8ea02804bb7aa1b9fd114/ESSP-survey.pdf

- DX + PipeDrive case study describing the process of improving developer productivity
  https://getdx.com/blog/pipedrive-developer-productivity/

- Realistic expectations regarding AI:
 https://www.thoughtworks.com/en-au/insights/blog/generative-ai/reckoning-generative-ai-uncanny-valley

Appendix: Engineering success antipatterns

Antipatterns are "common solutions to common problems where the solution is **ineffective and may result in undesired consequences.**" The first step in GitHub's ESSP is to understand friction or bottlenecks in the team or organization's engineering system. The following table provides examples of antipatterns that may be synonymous with friction or bottlenecks. The table also lists potential changes that may address the antipatterns and potential leading metrics and indicators that may be used to monitor whether the changes are having the desired impact. GitHub recommends asking engineering teams to support the identification of these antipatterns and to confirm the leading indicators that are best suited to the situation, as those listed may not be an appropriate fit for your situation.

| Anti-Pattern	Big Bang Releases
Description	Teams wait too long to release, deploying large batches of code at once.
Potential root-causes	Fear of destabilization with frequent releases. Lack of CI/CD pipeline maturity. Preference for 'all-at-once' (or quality) certainty. Strict compliance requirements. Long review cycles between PR and deployment.
Quality impact	Bugs and regressions are harder to detect and fix in larger code bases. Some features may also be released without having met quality expectations.
Velocity impact	Slows release cycles due to complex, high-risk deployments.
How AI could help	Use GitHub Copilot to write and review code faster, potentially leading to quicker PR completion, leading to more frequent deployments. Detect and resolve integration issues to prevent change failures.
Friction requiring non-AI intervention	Cultural issues or lack of communication between teams.
Potential leading or additional metrics or indicators that may indicate this antipattern [↑ ↓ trend suggestions antipattern]	Size of PRs ↑ PRs reviewed not merged ↑ PR review time ↑ Long-lived feature branches ↑
Zone metrics that may indicate this antipattern [↑ ↓ trend suggestions antipattern]	Deployment frequency ↓ Change failure rate ↑ Lead time ↑

Anti-Pattern	Gold Plating	Overengineering	Racking up technical debt
Description	Developers spend too much time perfecting code or adding unnecessary features.	Building overly complex solutions for simple problems.	Ignoring or deferring technical debt, allowing inefficient and vulnerable systems to persist.
Potential root-causes	Culture of perfectionism. Desire to showcase technical skills. No clear MVP focus or feature prioritization.	Desire to future-proof unnecessarily. Pressure to add value through complexity.	Deadline-driven focus on features. Long-term impact of technical debt undervalued. Significant risk in unknown upgrades and effort to resolve incompatibility issues.
Quality impact	Increased complexity introduces more potential for bugs without added value to user.	Complex systems are more prone to bugs and harder to maintain.	Code becomes brittle and bug-prone, leading to poor system health.
Velocity impact	Adds unnecessary time to development as teams over-focus on perfection.	Slows development as complexity adds overhead to build and maintain systems.	Increases time to develop new features as workarounds grow.
How AI could help	Use GitHub Copilot to simplify code and remove redundant code.	Use GitHub Copilot to refactor existing code. This could be to make the code more modular, or to suggest a simpler way of solving the problem.	Use GitHub Copilot to create tests and refactor existing code. This could be to make the code more modular, or to suggest a simpler way of solving the problem. Autofix may reduce effort and increase satisfaction with starter suggestions in PRs.
Friction requiring non-AI intervention	Product management decisions about feature prioritization.	Overdesigning systems to solve edge cases that rarely occur.	Prioritize and allocate engineers to address the technical debt.
Potential leading or additional metrics or indicators that may indicate this antipattern [↑↓ trend suggestions antipattern]	Work in Progress ↑ Late-in cycle code churn ↑ Usage of features/sub-features ↓	Developer satisfaction with delivery cadence ↓ Usage of features/sub-features ↓ Cognitive complexity ↑	Code complexity ↑ Large blocks of commented out code ↑ Duplicated Blocks ↑ Hardcoded values and secrets ↑ Dependency issues ↑
Zone metrics that may indicate this antipattern [↑↓ trend suggestions antipattern]	Lead time ↑	Code security and maintainability ↓ Lead time ↑	Code security and maintainability ↓ Lead time ↑ Change Failure Rate ↑

Anti-Pattern	Unclear requirements	Manual deployments	Testing bottlenecks
Description	Teams receive vague or incomplete requirements, leading to misunderstandings.	Relying on manual steps for deployment instead of automation.	Relying on manual testing or insufficient test automation, causing delays.
Potential root-causes	Pressure to start development quickly. Immature product discovery process. Frequent priority shifts from stakeholders.	Perception that manual is 'good enough.' Fear of effort needed for automation. Lack of investment in DevOps practices.	Belief in thoroughness of manual testing. Resource constraints for automation. Limited familiarity with modern test tools. Previous experience with brittle, costly, or flaky tests.
Quality impact	Poorly defined requirements lead to incorrect or low-quality features.	Manual deployments introduce inconsistent outcomes that can lead to post-deployment bugs.	Lack of thorough testing introduces more bugs into production.
Velocity impact	Time wasted clarifying requirements or building incorrect features.	Slows releases.	Delays releases as testing takes longer.
How AI could help	Stay tuned: GitHub's AI powered platform continues to evolve	Use GitHub Copilot to create automation, such as GitHub Action workflows, to replace manual deployments. Use GitHub Copilot to troubleshoot why a deployment automation has failed.	Use GitHub Copilot to create test suites, and automate CI workflows, to remove frictions.
Friction requiring non-AI intervention	Engaging with stakeholders to ensure real-world needs are reflected in the requirements.	Inconsistent processes and human reluctance to adopt automated deployment pipelines.	The need for a robust testing strategy aligned with the project's goals
Potential leading or additional metrics or indicators that may indicate this antipattern [↑↓ trend suggestions antipattern]	Time spent in meetings ↑ Work in Progress ↑ Rework ↑ Developer frustration ↑	Count of manual steps per deployment ↑ Dwell (delay) time during CI/CD ↑ Deployment duration ↑	Automated test coverage ↓ Time spent on manual testing ↑
Zone metrics that may indicate this antipattern [↑↓ trend suggestions antipattern]	Flow state experience ↓ Lead time ↑ PRs merged per developer ↓	Deployment frequency ↓ Failed deployment recovery time ↑ Change failure rate ↑ Engineering tooling satisfaction ↓	Change failure rate ↑ Deployment frequency ↓ (Median) Lead time ↑ Engineering tooling satisfaction ↓

Anti-Pattern	Siloed teams	Inconsistent feedback loops	Scope creep
Description	Teams operate in silos, failing to share data, tools, or processes across teams.	Feedback from testing, users, or other stakeholders is not provided in a timely or consistent manner.	Constant addition of features or changes mid-development without proper evaluation.
Potential root-causes	Incentives misaligned across teams. Culture prioritizes team-specific goals. Historical habit of independent operation.	Waterfall mindset undervaluing iteration. Feedback viewed as an end-phase activity. Lack of real-time feedback tools.	Unclear project boundaries. Poor change management practices. Culture discourages saying "no" to requests.
Quality impact	Inconsistent processes and tools result in lower-quality handoffs between teams.	Bugs and user issues linger due to delayed feedback.	Rushed development due to scope creep often leads to more bugs and lower quality.
Velocity impact	Cross-team dependencies lead to delays when teams aren't aligned.	Slows iteration cycles, as engineers aren't able to adapt quickly.	Introduces unplanned work that delays original timelines.
How AI could help	Copilot features can help improve documentation and code explanations.	Use GitHub Copilot for Pull Requests to automatically analyze pull requests and suggest changes to provide a more consistent feedback loop. Developers can use GitHub Copilot to ask questions about a pull request, providing for a more informed pull request review that leads to a more consistent feedback loop	Stay tuned: GitHub's AI powered platform continues to evolve
Friction requiring non-AI intervention	Cultural issues or lack of communication between teams.	Human communication and prioritization of feedback.	Managing stakeholder expectations and ensuring a disciplined approach to scope management.
Potential leading or additional metrics or indicators that may indicate this antipattern [↑ ↓ trend suggestions antipattern]	Cross-team collaboration frequency ↓ Handoff delays ↑ Rework frequency ↑ Poor meeting attendance ↑	Feedback frequency ↓ Feedback quality ↓ Customer satisfaction ↓ Age of PR's last human activity ↑	Scope changes per sprint ↑ Ratio of issue types per sprint ↑ Time spent on unplanned work ↑
Zone metrics that may indicate this antipattern [↑ ↓ trend suggestions antipattern]	Lead time ↑ PRs merged per developer ↓ Deployment frequency ↓	Lead time ↑ Flow state experience ↓	Lead time ↑