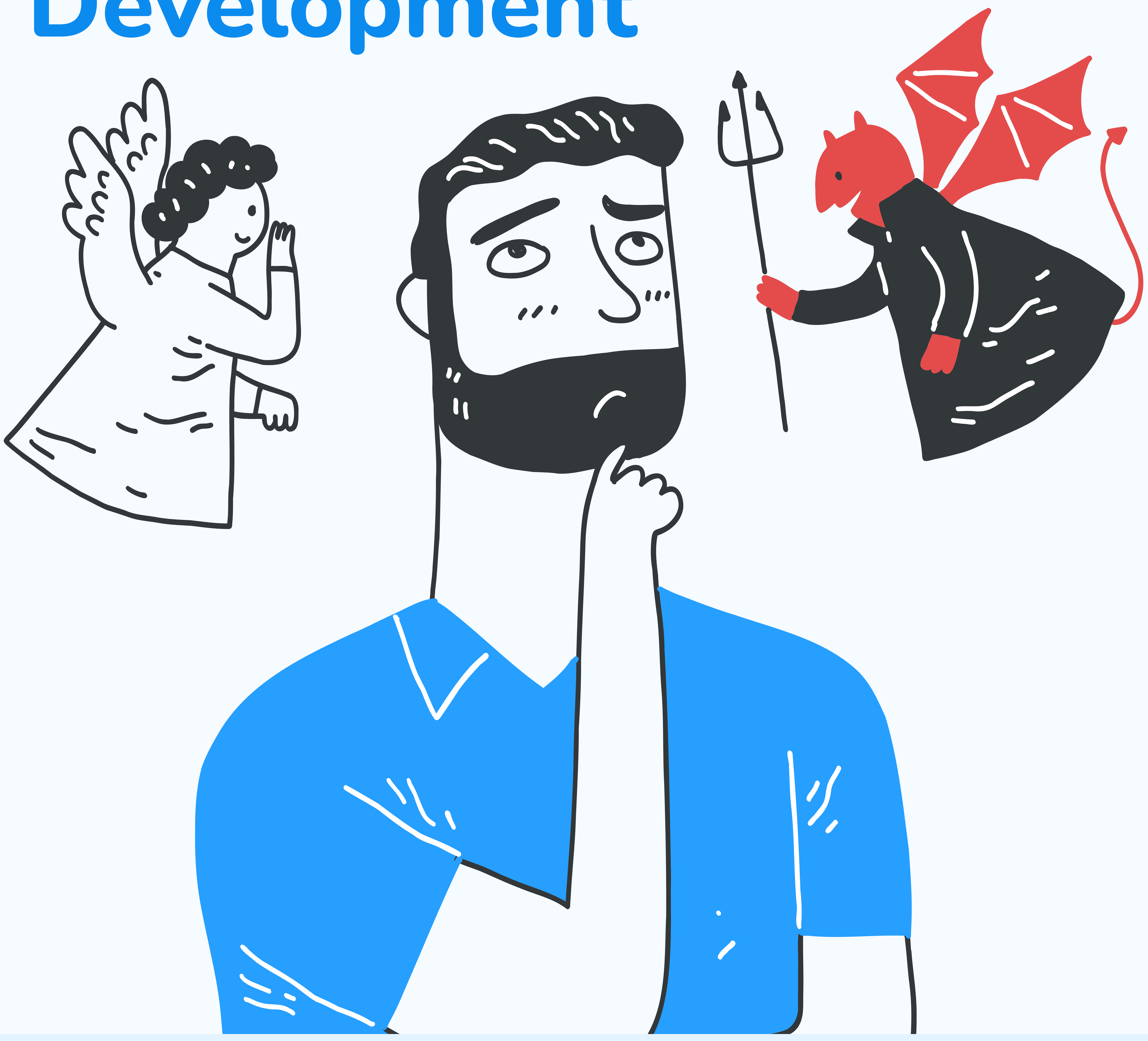
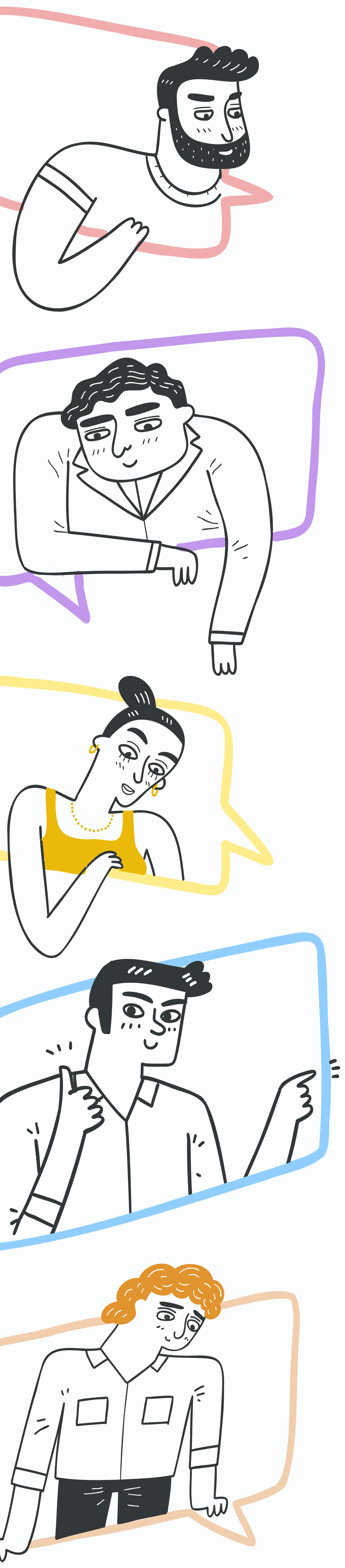
The Good, the Bad, and the Ugly: Exploring 26 Behavioral Practices in Software Development



A handbook for understanding your software engineering team

hay



Introduction

Though I'm quite sure the list could be longer depending on who's creating it, my engineers and I have attempted to distill the most crucial behaviors in technology development.

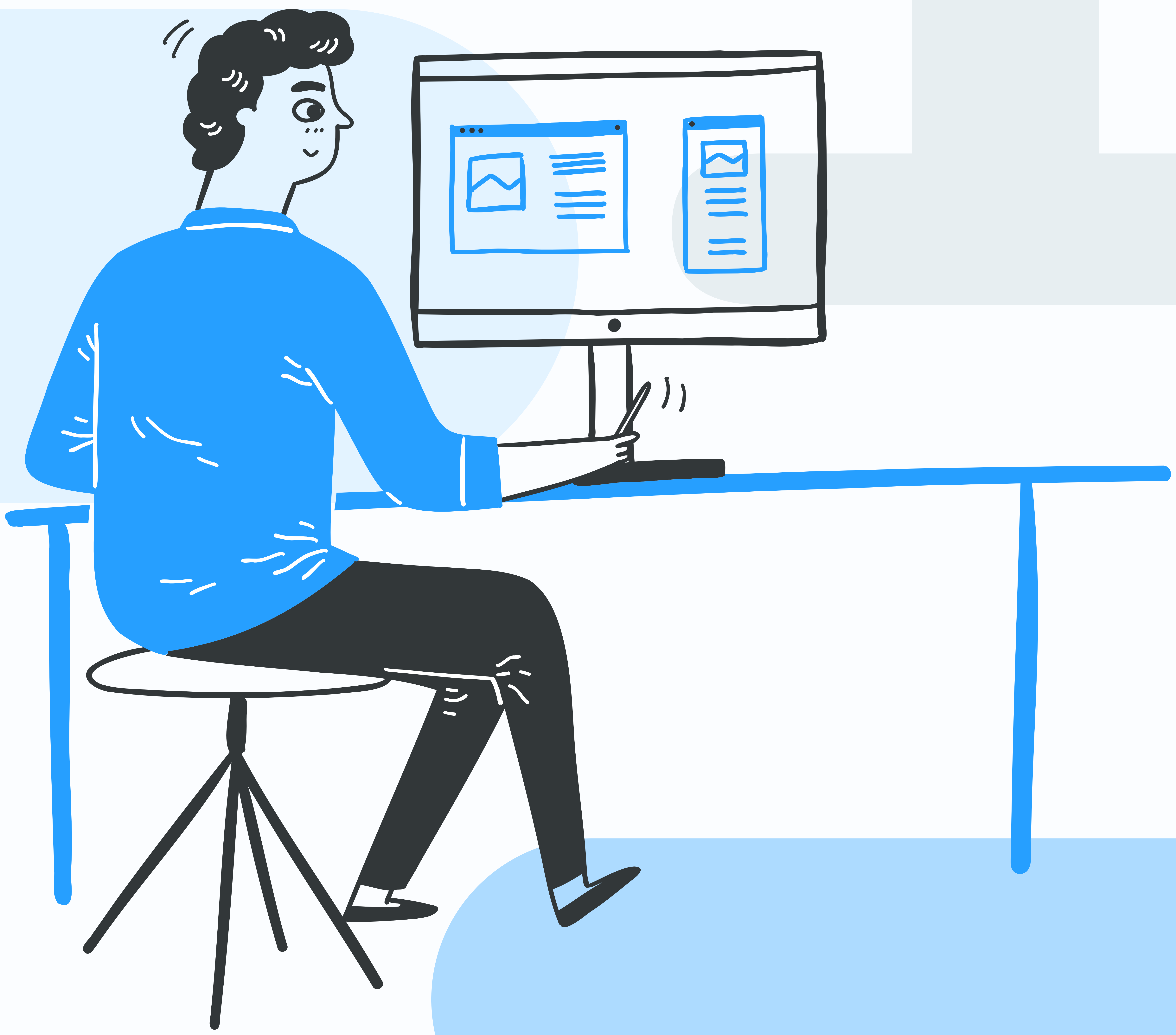
Patterns, trends, or behaviors – regardless of your choice of terminology, one thing remains clear: some of them can substantially impede your development team's success, while others can significantly bolster it.

By exploring these **patterns at the individual, team, and leadership levels**, you'll gain a deeper understanding of your role as a manager, the dynamics of your development team, and how your individual contributors perform at their best (or worst). Furthermore, deciding which ones are genuinely 'ugly' or 'bad' may be as subjective as your personal judgment.

I hope you'll find our guide on behavioral practices in coding insightful. Feel free to share it among your dev teams!

Tomasz Korzeniowski

Developer Behaviors



1

🚨 Putting Out Fires

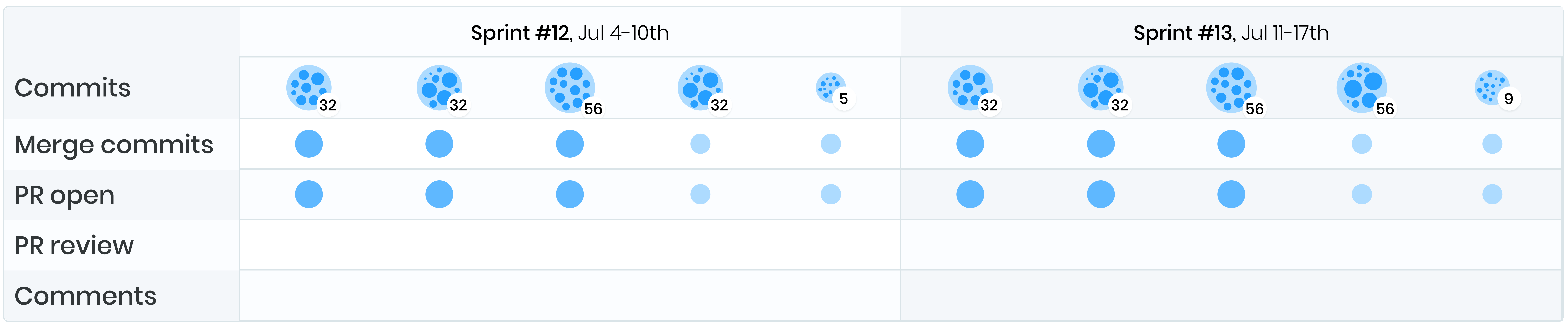
Sometimes a reviewer fixes a bug or a critical defect at the eleventh hour. This pattern, often seen right before a release, can foster a reliance **on last-minute heroics to resolve issues** that could have been addressed earlier, leading to a stressful and unsustainable work environment.

How to recognize it:

- **Frequent Last-Minute Commits:** Look for significant code changes just before deadlines, indicating procrastination and rushed work.
- **High Stress Pre-Release:** Notice increased tension and overtime near release deadlines, suggesting hurried fixes and poor planning.
- **Reliance on Specific Developers:** Observe if a few team members repeatedly handle urgent last-minute issues, indicating an imbalanced skill distribution.
- **Limited Early Review Engagement:** Check for superficial early code reviews, leading to missed early problem detection.
- **Late Bug Discovery:** Monitor for bugs often found and fixed too close to deadlines, showing insufficient early testing.
- **Hesitation for Early Feedback:** Be aware of developers delaying sharing work or seeking feedback, potentially leading to bigger issues later.

How to discourage it:

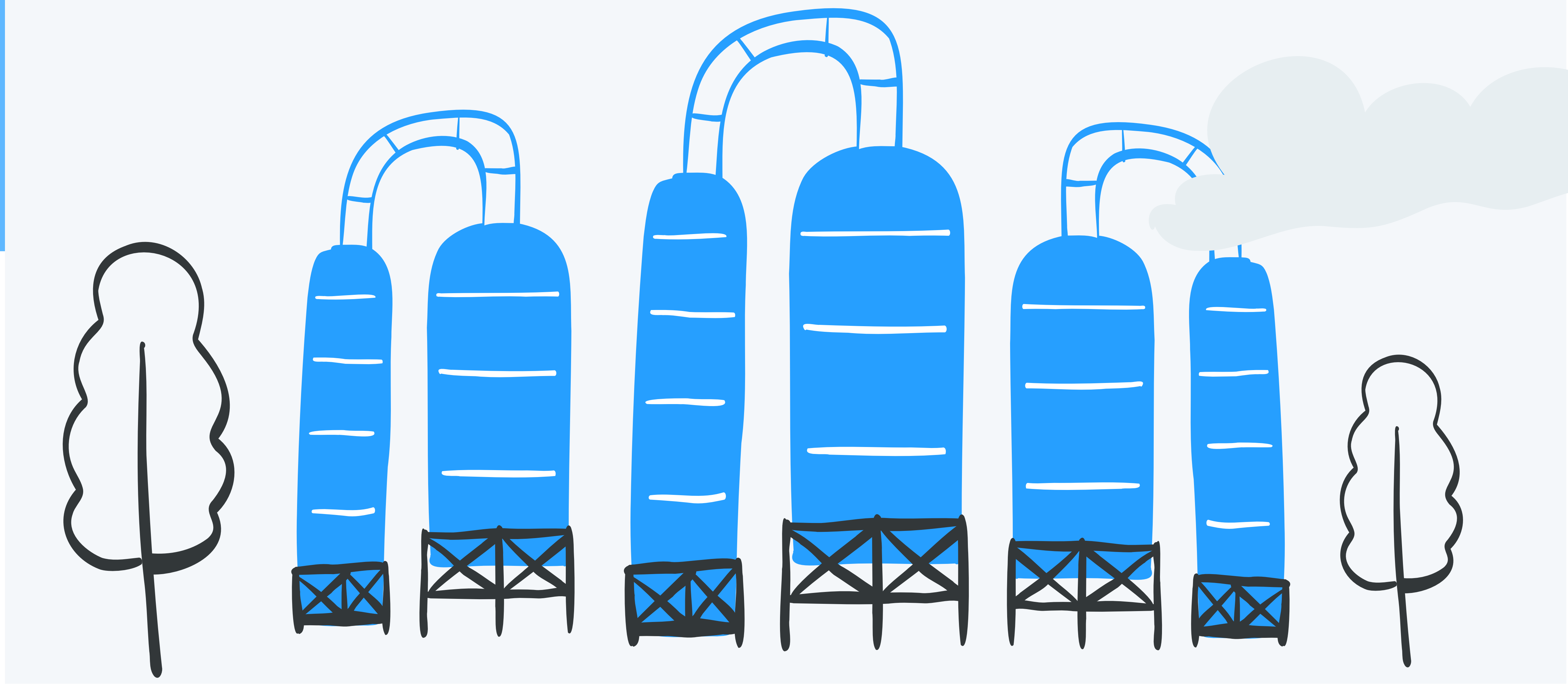
- **Establish Regular Review Cycles:** Implement consistent and systematic, incremental code reviews to catch issues earlier in the process.
- **Promote a Balanced Workload:** Actively manage and distribute tasks evenly to prevent over-reliance on specific team members.
- **Foster a Culture of Early Feedback:** Encourage the norm of seeking and providing constructive feedback early in the development cycle.
- **Comprehensive Review Training:** Provide detailed training sessions focused on enhancing code review techniques and effective communication skills.
- **Set Clear Expectations and Deadlines:** Clearly outline and communicate project timelines and objectives, ensuring team alignment and understanding.
- **Utilize Predictive Analytics Tools:** Employ advanced tools designed to predict and address potential bottlenecks in the software development process.



	Sprint #12, Jul 4-10th					**Sprint #13**, Jul 11-17th				
Commits	32	32	56	32	5	32	32	56	56	9
Merge commits	●	●	●	●	●	●	●	●	●	●
PR open	●	●	●	●	●	●	●	●	●	●
PR review										
Comments										

Repository /hay-backend

Lack of code reviews and fixing bugs at the end of the sprint



2 Coding in Silos

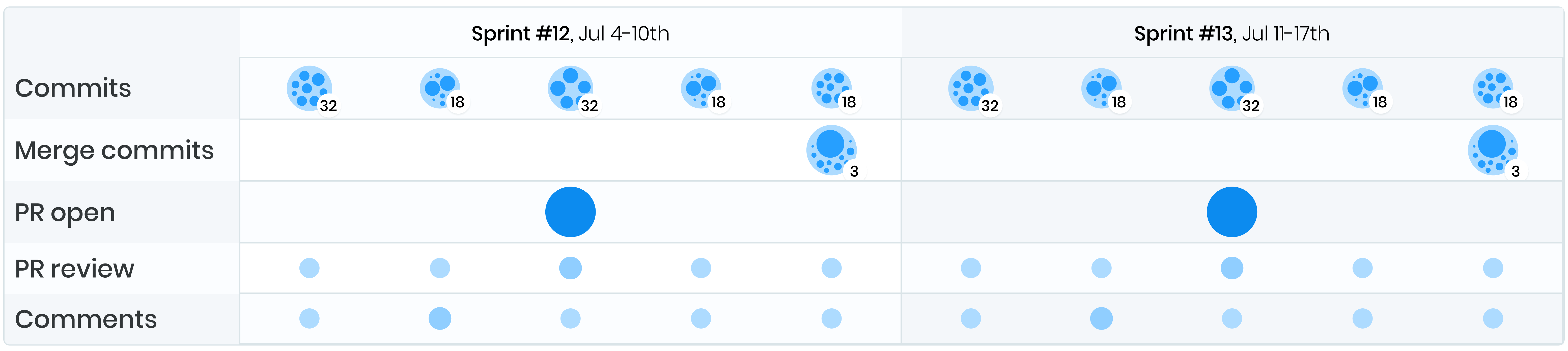
Engineering Managers should discourage "coding in silos," characterized by developers **working in isolation** and submitting large PRs at the sprint's end. This pattern hampers teamwork and causes bottlenecks. Encouraging smaller, well-planned PRs helps avoid these issues and ensures smoother teamwork.

How to recognize it:

- **Large, Infrequent Commits:** Watch for sparse activity followed by big commits, suggesting isolated, bulk coding.
- **Large Pull/Merge Requests:** Note unusually large PRs/MRs, typically submitted late, indicating batch code dumping.
- **Fewer, Bigger Commits/PRs:** Smaller count of larger commits/PRs may point to a code hoarding habit.
- **Reduced Comment Activity:** Sparse comments with late-sprint spikes could indicate minimal ongoing collaboration.
- **Pattern Consistency:** Repeated large, last-minute commits across sprints signal a habitual code hoarding practice.
- **PR/MR Merge Delays:** Lengthy review and merge times for large PRs/MRs, often extending into next sprints.

How to discourage it:

- **Foster Incremental Development:** Encourage developers to break down their work into smaller, manageable tasks, promoting regular integration and completion.
- **Implement Mandatory Code Reviews:** Establish a process where every commit must be reviewed before merging, leading to more frequent, manageable commits.
- **Promote Pair Programming:** Pair programming improves code quality, ensures continuous knowledge sharing, and encourages collaborative development.
- **Introduce Continuous Integration (CI) Practices:** Implement CI tools that automatically build and test code with every commit, ensuring early error detection.
- **Educate on the Risks of Large PRs:** Share insights on the challenges of reviewing large PRs, emphasizing increased error risk and team stress.
- **Enforce Early Feedback Loops:** Create a culture where seeking early and frequent feedback is the norm, fostering proactive problem-solving and collaboration.



Repository /hay-backend

3 ☘️ Precision in Code Submissions

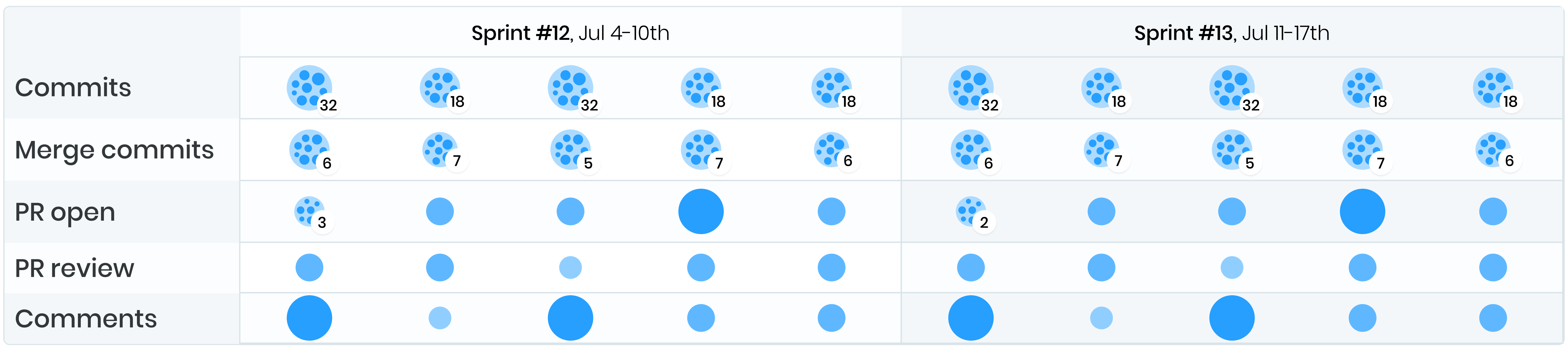
Engineers often break down **complex issues into manageable tasks**. They then submit precise, high-quality code that often requires little to no revision. While beneficial, this meticulous work can go unrecognized within teams.

How to recognize it:

- **Low Revision Rate:** Rarely need significant post-review changes.
- **High First-Time Acceptance:** PRs/MRs often approved in the first review.
- **Minimal Bugs or Issues:** Code usually has minimal testing and production issues.
- **Code Quality Metrics:** Tools that measure code quality (like static code analyzers) score highly on submissions from these engineers.
- **Peer Feedback:** Positive feedback and fewer requests for changes or clarifications from peers during the code review process.
- **Documentation and Code Comments:** Clear, concise, and useful documentation and comments.

How to encourage it:

- **Cultivate a Recognition-Rich Environment:** Regularly spotlight and celebrate precise coding in meetings.
- **Incentivize Thoroughness:** Reward clean and precise code with incentives like gamification or tangible rewards.
- **Integrate Precision into Performance Metrics:** Make code precision a key factor in performance reviews.
- **Promote Continuous Learning:** Provide ongoing training to enhance coding best practices.
- **Provide the Right Tools:** Equip teams with sophisticated tools for precise coding, including IDEs and linters.
- **Foster Mentorship:** Connect novice developers with mentors to refine coding precision.



	Sprint #12, Jul 4-10th					**Sprint #13**, Jul 11-17th				
Commits	32	18	32	18	18	32	18	32	18	18
Merge commits	6	7	5	7	6	6	7	5	7	6
PR open	3	●	●	●	●	2	●	●	●	●
PR review	●	●	●	●	●	●	●	●	●	●
Comments	●	●	●	●	●	●	●	●	●	●

Repository /hay-backend

Consistent workflow along with code reviews

I'm not a great programmer; I'm just a good programmer with great habits. — Kent Beck

4

🍀 Entering Flow

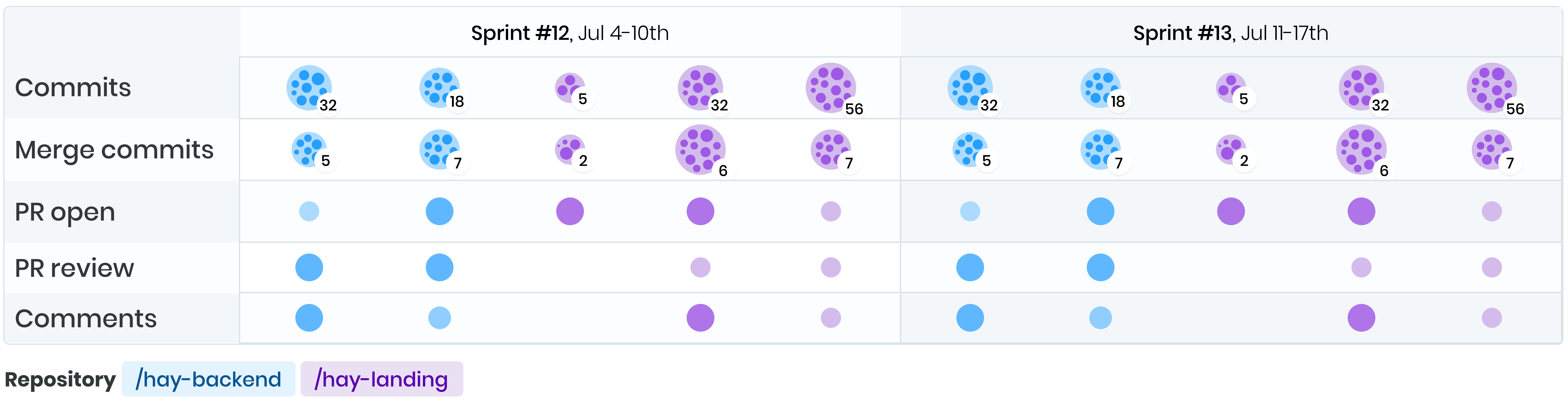
Flow is characterized by engineers who consistently **produce focused, reliable, and high-quality work**. They excel at maintaining a state of deep concentration, often referred to as being 'in a state of flow' which results in efficient and effective output.

How to recognize it:

- **Consistent Productivity:** Engineers demonstrate a steady output of work, with commits and PRs showing a regular and sustained pattern, indicating prolonged periods of concentration.
- **High-Quality Work:** The code submitted has fewer bugs or issues, and there's a notable reduction in the number of revisions required.
- **Long Uninterrupted Periods:** Time tracking or self-reporting shows periods where the engineer is working without breaks.
- **Engagement in the Zone:** The engineer is less responsive to external communications during certain periods, signaling that they are deeply engaged in their tasks.
- **Minimal Multi-Tasking:** There is a clear focus on single tasks at a time.
- **Reduced Context Switching:** A decrease in the frequency of switching between different projects or tasks.
- **High Task Completion Rate:** Tasks are not only started but also seen through to completion without stalling or being carried over to subsequent sprints.

How to encourage it:

- **Optimize Meeting Schedules:** Hold meetings early or in blocks to free up uninterrupted time for focused work.
- **Designate Quiet Hours:** Set "Do Not Disturb" periods during high productivity hours for concentrated deep work.
- **Establish No-Meeting Days:** Set specific days without meetings to allow for extended, uninterrupted coding.
- **Encourage Time Blocking:** Support scheduling dedicated coding periods, free from meetings and interruptions.
- **Foster Shared Ownership:** Distribute tasks evenly to prevent bottlenecks and reliance on single team members.
- **Balance Workloads:** Ensure workloads are manageable, allowing deep focus without overburdening developers.
- **Acknowledge Individual Work Styles:** Respect and adapt to each developer's unique productivity times and preferences.
- **Leverage Productivity Tools:** Use tools to reduce distractions and enhance focus.



Context switching as an example of breaking flow

5

🚨 Excessive Rework

Effective management of rework in coding, which includes testing, refining code, and exploring various solutions, involves understanding the typical rework level for your team across different project stages and individual workflows. Excessive rework, however, indicates potential issues such as **unclear stakeholder requirements**, perfectionism, or difficulties with certain code aspects.

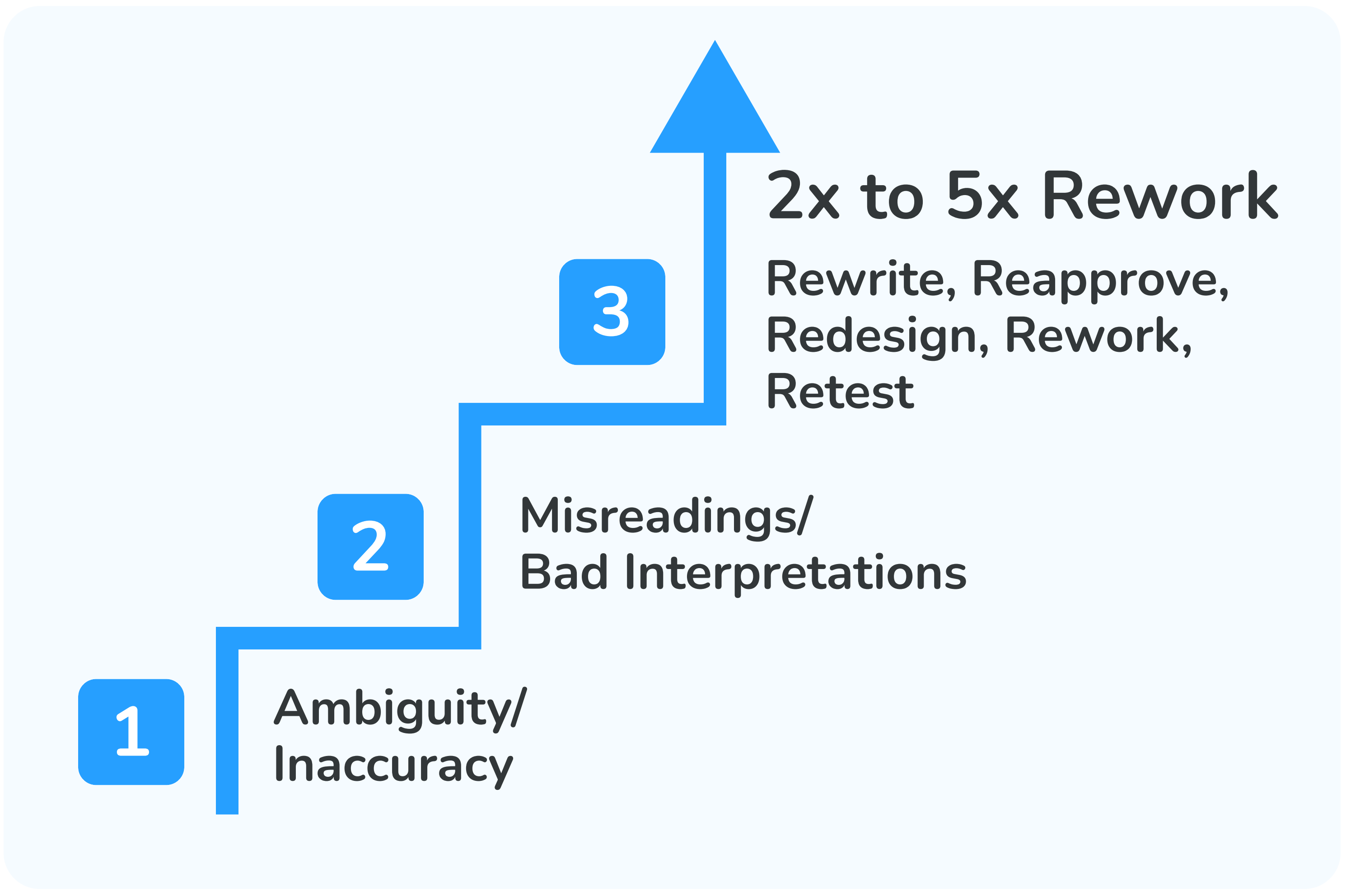
How to recognize it:

- **High Commit Frequency:** Regular commits to same code areas indicate constant need for revisions.
- **Long PR Times:** Prolonged pull request durations suggest ongoing adjustments and iterative changes.
- **Increased Code Churn:** Rapid addition and removal of code reflect ongoing instability and corrections.
- **Review Pushbacks:** Frequent requests for significant changes during reviews signal extensive rework.
- **Repeated Review Themes:** Ongoing issues in code reviews highlight continuous coding challenges.
- **Missed Deadlines:** Consistently exceeding deadlines indicates continual rework and project time overruns.
- **Rising Bug Counts in Subsequent Tests:** Code changes that result in new bugs or failures in areas that previously passed testing can signal inefficient rework.
- **Frustration or Fatigue:** Repetitive rework often leads to developer stress and weariness.
- **Frequent Peer Consultation:** Regularly seeking validation from peers suggests reliance and uncertainty.

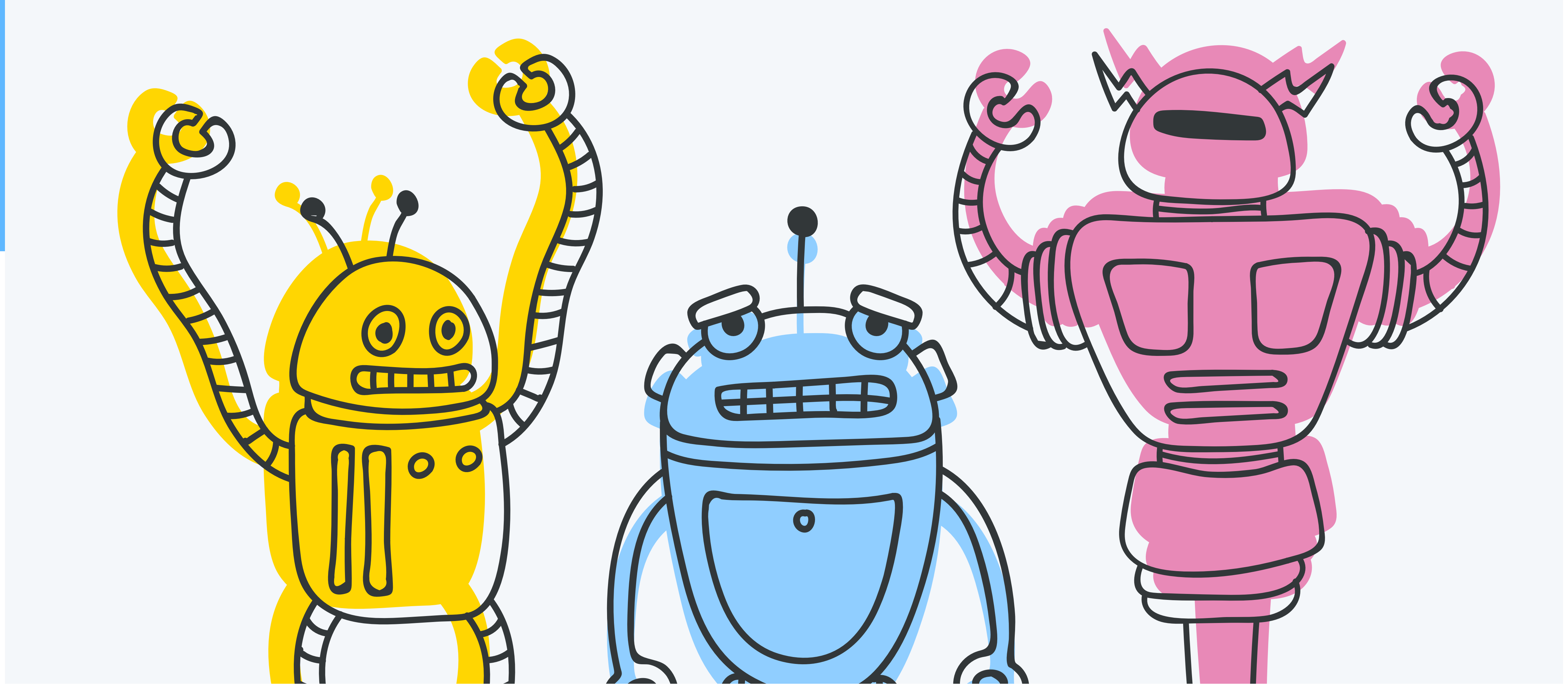
This is what makes them good engineers. Perfectionism: incinerating perfectionism.
— Ellen Ullman, The Bug

How to discourage it:

- **Initiate Open Conversations:** Engage in honest discussions to determine the reasons behind frequent rework.
- **Document and Analyze Impact:** Track and present the time and impact of rework versus new development.
- **Set Quality Benchmarks:** Set and communicate "good enough" standards in collaboration with senior engineers.
- **Facilitate Collaborative Coding:** Use pair programming for complex tasks to reduce rework needs.
- **Prioritize Pre-Review Checks:** Conduct informal peer reviews before formal processes to identify issues early.
- **Leverage Expertise:** Involve domain experts for developers struggling with unfamiliar areas.
- **Adjust Workflows:** Modify development workflows to allow comprehensive task understanding.


2x to 5x Rework
3
Rewrite, Reapprove, Redesign, Rework, Retest
2
Misreadings/ Bad Interpretations
1
Ambiguity/ Inaccuracy

Poorly defined requirements can be a cause of excessive rework.
https://www.modernrequirements.com/blogs/medical-device-software-development/



6

🚨 Overusing AI Assistants

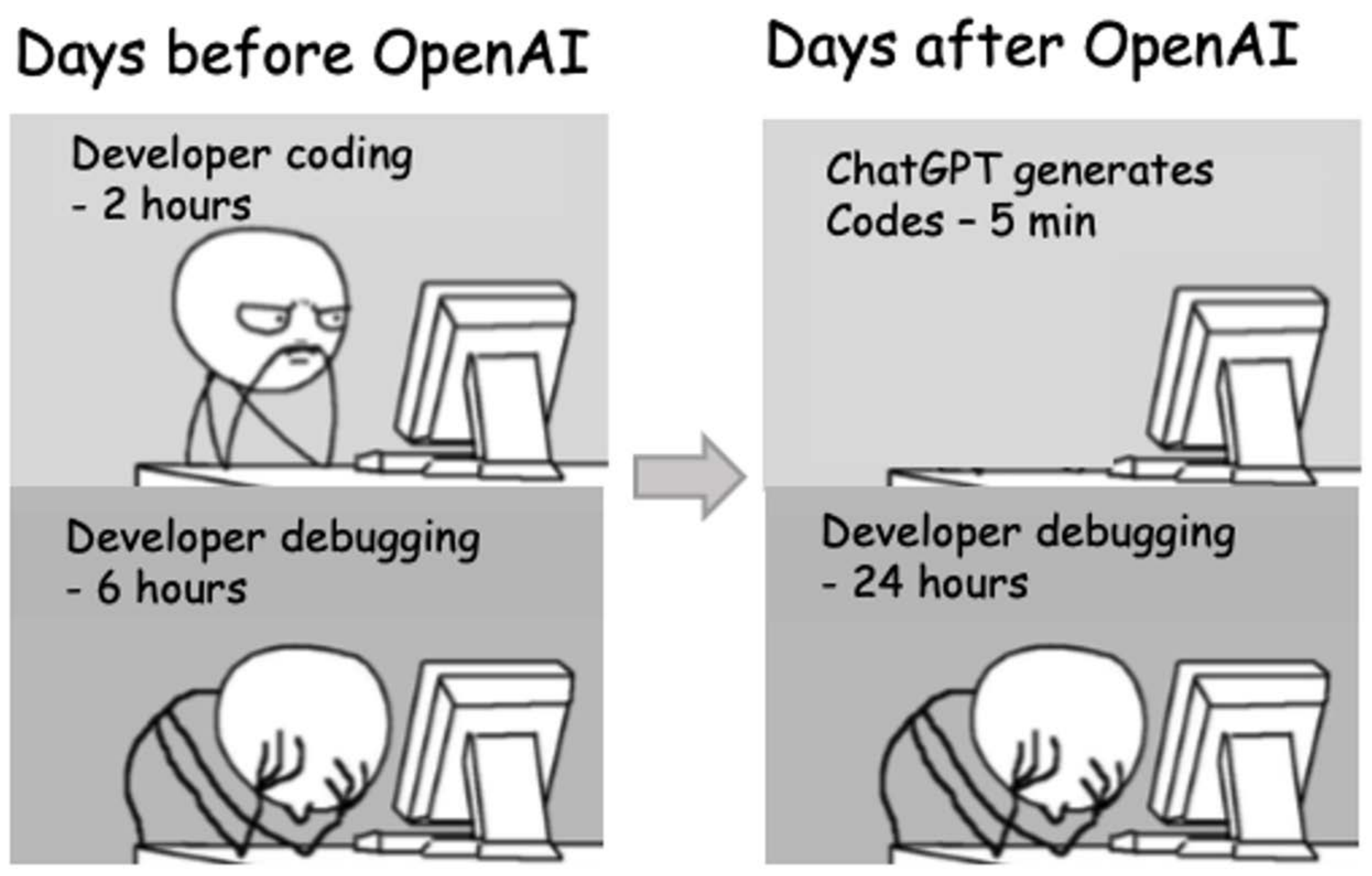
While using AI coding assistants can increase **efficiency and productivity** 🤖, overreliance can limit capacities and stifle developer creativity. AI assistants can also lead to instances of biased and inaccurate outputs, as evidenced by studies showing a significant percentage of incorrect responses and code generation by such tools.

How to recognize it:

- **Overdependence on AI:** Excessive use of AI for complex tasks where human judgment is key.
- **Quality Drop in Human Tasks:** Declining quality in work areas that require nuanced human skills.
- **Less Team Engagement:** Reduced enthusiasm in tasks as AI takes over human-centric roles.
- **AI-Related Errors:** Increase in mistakes due to inappropriate or excessive AI usage.
- **Stalled Skill Growth:** Noticeable decline in team's skill development due to reliance on AI.
- **Diminished Problem-Solving Skills:** Decreased ability in the team to solve complex problems without AI assistance.
- **Reduced Creative Output:** A noticeable decrease in innovative and creative solutions as AI-driven answers become the norm.

How to discourage it:

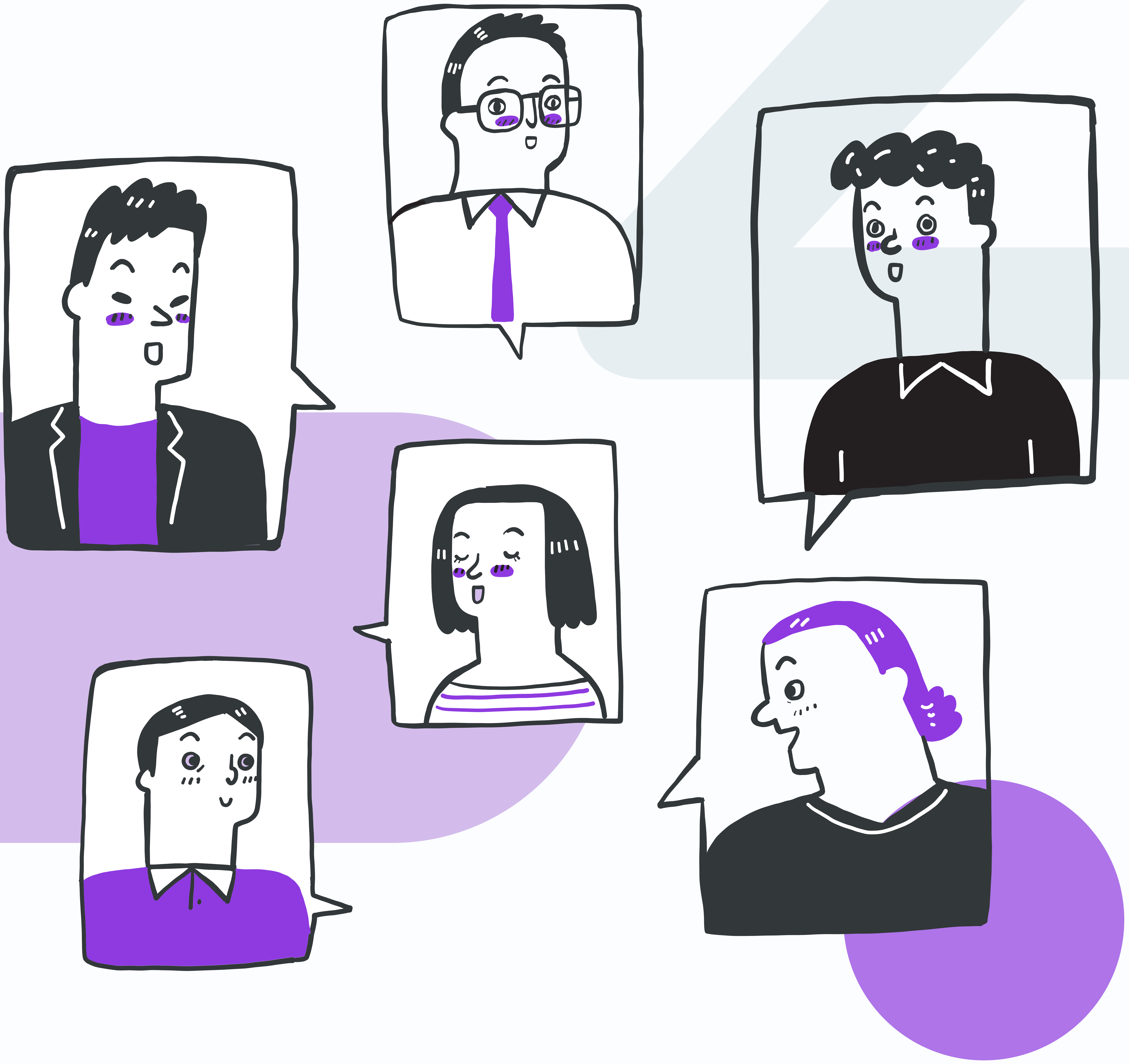
- **Establish Clear AI Usage Guidelines:** Define specific scenarios where AI assistance is appropriate.
- **Promote Understanding of AI Limitations:** Educate developers on the limitations and potential biases of AI.
- **Encourage Manual Problem-Solving Skills:** Foster an environment where developers are encouraged to tackle problems manually first.
- **Monitor AI Dependence:** Regularly review projects to ensure a balanced use of AI.
- **Highlight Human-Centric Design:** Stress the importance of human-centric design in software development, where AI is used to enhance user experience and functionality, not dictate the development process.
- **Regular Skill Development Sessions:** Organize workshops and training focusing on enhancing core programming and design skills, reducing over-reliance on AI.


Days before OpenAI
Developer coding - 2 hours
Developer debugging - 6 hours
Days after OpenAI
ChatGPT generates Codes - 5 min
Developer debugging - 24 hours

https://www.reddit.com/r/meme/comments/14pm80n/chatgpt_is_actually_a_problem_for_developers/

For all the hype about not having to write programs, we cannot forget that any programmer, human or automatic, needs specifications, and that any candidate program requires verification.
— Bertrand Meyer

2 Team Behaviors



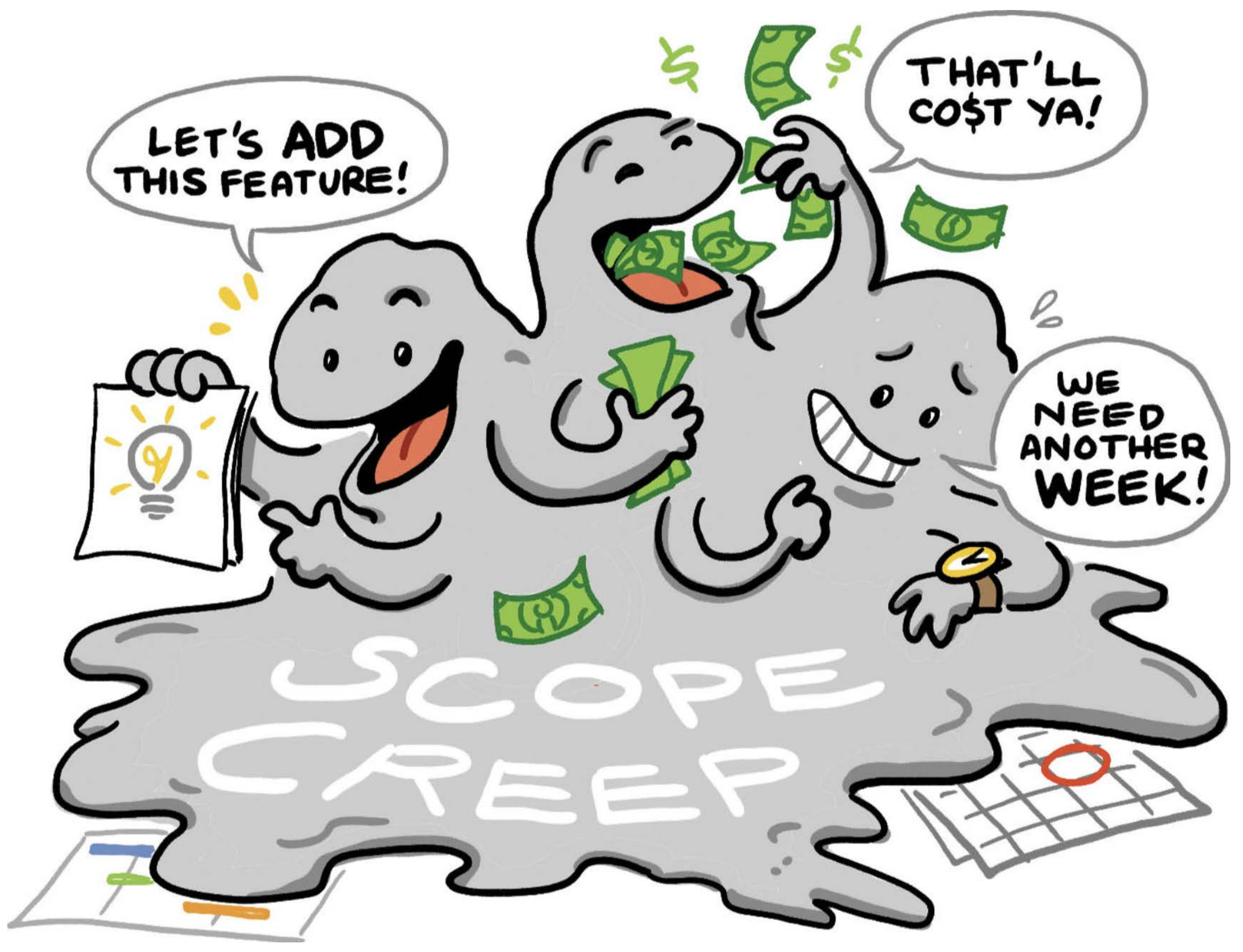
7

🚨 Scope Creep

Scope Creep is a common problem where the boundaries of a project expand beyond the original plans after the work has started. **This leads to incremental increases in workload and complexity.** It typically stems from either a product owner overly focused on aligning with stakeholders while disregarding the team's feedback about unrealistic expectations or quality concerns, or stakeholders directly approaching developers, persuading them to quickly **implement "just this one thing"** outside the normal workflow.

How to recognize it:

- **Unexpected Task Addition:** Notice a rise in unplanned tasks or features being added after project kickoff.
- **Frequent Requirement Changes:** Regular changes to project requirements, often beyond initial project scope.
- **Deadline Extensions:** Consistent extensions of project deadlines due to added workload.
- **Developer Overload:** Developers express concerns about increased workload or complexity beyond original plans.
- **Stakeholder Pressure:** Increased instances of stakeholders requesting changes or additions directly from developers.
- **Reduced Quality Focus:** Shift from quality to quantity in deliverables as project scope expands.



https://www.flickr.com/photos/rosenfeldmedia/35473616510

How to discourage it:

- **Clear Scope Definition:** Establish and adhere to a well-defined project scope from the outset.
- **Stakeholder Communication:** Ensure open and regular communication channels between stakeholders and project managers.
- **Change Management Process:** Implement a structured process for evaluating and approving scope changes.
- **Developer Advocacy:** Encourage product owners to prioritize team feedback about workload and feasibility.
- **Regular Scope Reviews:** Conduct periodic reviews to assess the scope against initial objectives and resource allocation.
- **Educate on Impact:** Highlight the negative impact of scope creep on project timelines, budgets, and team morale.
- **Firm Boundaries:** Set and enforce strict boundaries for stakeholder requests, especially direct developer interactions.

"*It is always easier to talk about change than to make it.*"
— Alvin Toffler

8

😬 Working in Isolation

Working in isolation occurs when information isn't shared within specific team members or groups, leading to unnecessary friction and communication breakdowns. An example of this can be found in code reviews, where one reviewer reviews the code of only one other person in the team. **This can hinder the collaborative process** and result in only a few team members being equipped to conduct thorough code reviews, provide feedback, or troubleshoot issues effectively.

How to recognize it:

- **Limited Reviewers:** Consistently the same pair of team members involved in code reviews.
- **Knowledge Silos:** Certain team members hold exclusive knowledge, rarely shared with others.
- **Lack of Collaboration:** Minimal joint problem-solving or brainstorming sessions among team members.
- **Communication Gaps:** Noticeable lack of discussions or updates from specific team members.
- **Uneven Skill Distribution:** Skills and expertise concentrated in a few individuals, not spread across the team.
- **Isolated Task Completion:** Tasks or projects completed with little to no input from other team members.

https://sloanreview.mit.edu/article/solving-the-problem-of-siloed-it-in-organizations/

How to discourage it:

- **Rotate Reviewers:** Systematically change code reviewers to ensure broad participation and knowledge sharing.
- **Foster Knowledge Sharing:** Implement regular sessions for knowledge transfer and skill-building across the team.
- **Encourage Group Problem-Solving:** Regularly organize team brainstorming and collaborative problem-solving sessions.
- **Open Communication Channels:** Establish and maintain open lines of communication for all team members.
- **Team-Based Goals:** Set objectives that require collective input and collaborative effort to achieve.
- **Peer Mentorship Programs:** Pair less experienced team members with mentors for regular guidance and feedback.

"A dynamic duo who work well together can be worth any three people working in isolation."
— Larry Constantine

9 🚨 Rubber Stamping PRs

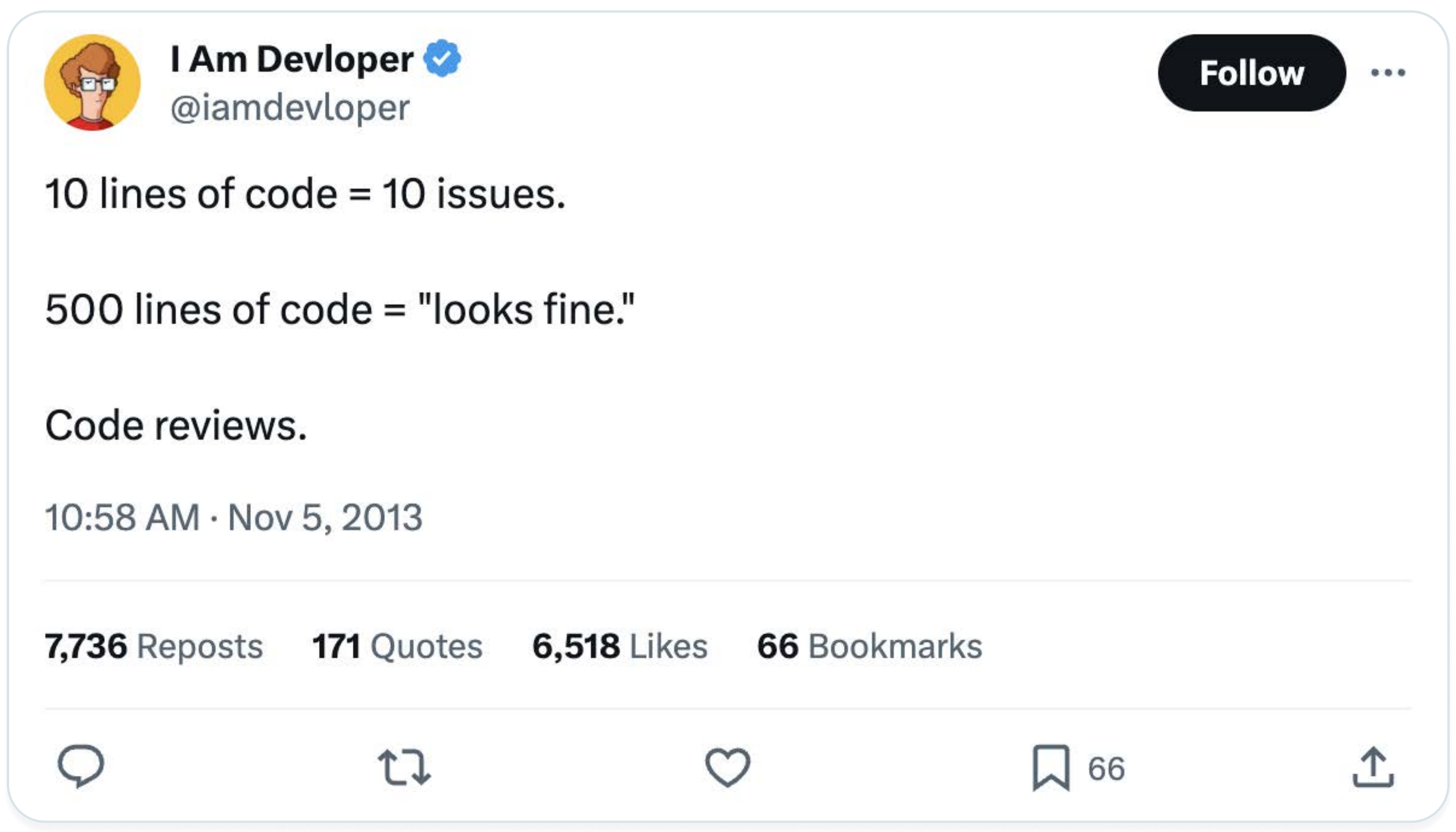
Rubber stamping is a situation where an engineer approves a peer's Pull Request without conducting a thorough and critical review. This behavior typically arises from an assumption that the submitted work inherently meets the necessary quality standards, possibly due to the submitter's experience or reputation. It may also be driven by external time pressures, such as looming deadlines or a high volume of pending reviews, which can lead to a cursory glance rather than an in-depth analysis of the code.

How to recognize it:

- **Quick Approvals:** Noticeably rapid PR approvals that suggest insufficient time for thorough review.
- **Lack of Detailed Feedback:** Minimal or non-specific feedback on PRs, indicating superficial reviews.
- **Seniority Assumptions:** Junior engineers frequently approving senior engineers' PRs without substantial comments.
- **Consistent Approver Patterns:** Same team members routinely approving each other's PRs with little scrutiny.
- **High Approval Rate:** An unusually high number of PRs approved on the first submission.
- **Limited Discussion Threads:** Sparse or non-existent discussion threads in the PR review section.
- **Repeat Issues Post-Merge:** Recurrent problems or bugs in code post-PR approval hinting at lax reviews.

How to discourage it:

- **Enforce Review Standards:** Establish and enforce clear guidelines for what constitutes a thorough PR review.
- **Regular Review Training:** Conduct training sessions on effective code review strategies and best practices.
- **Review Transparency:** Encourage openness in the review process, making all feedback visible to the entire team.
- **Time Allocation for Reviews:** Ensure adequate time is allocated for engineers to conduct proper PR reviews.
- **Peer Accountability:** Promote a culture of accountability where engineers are responsible for quality reviews.
- **Automated Review Checks:** Implement automated checks to flag PRs that might need more in-depth review.
- **Random Review Audits:** Periodically audit PR reviews to ensure compliance with established standards.



I Am Devloper @iamdevloper

10 lines of code = 10 issues.

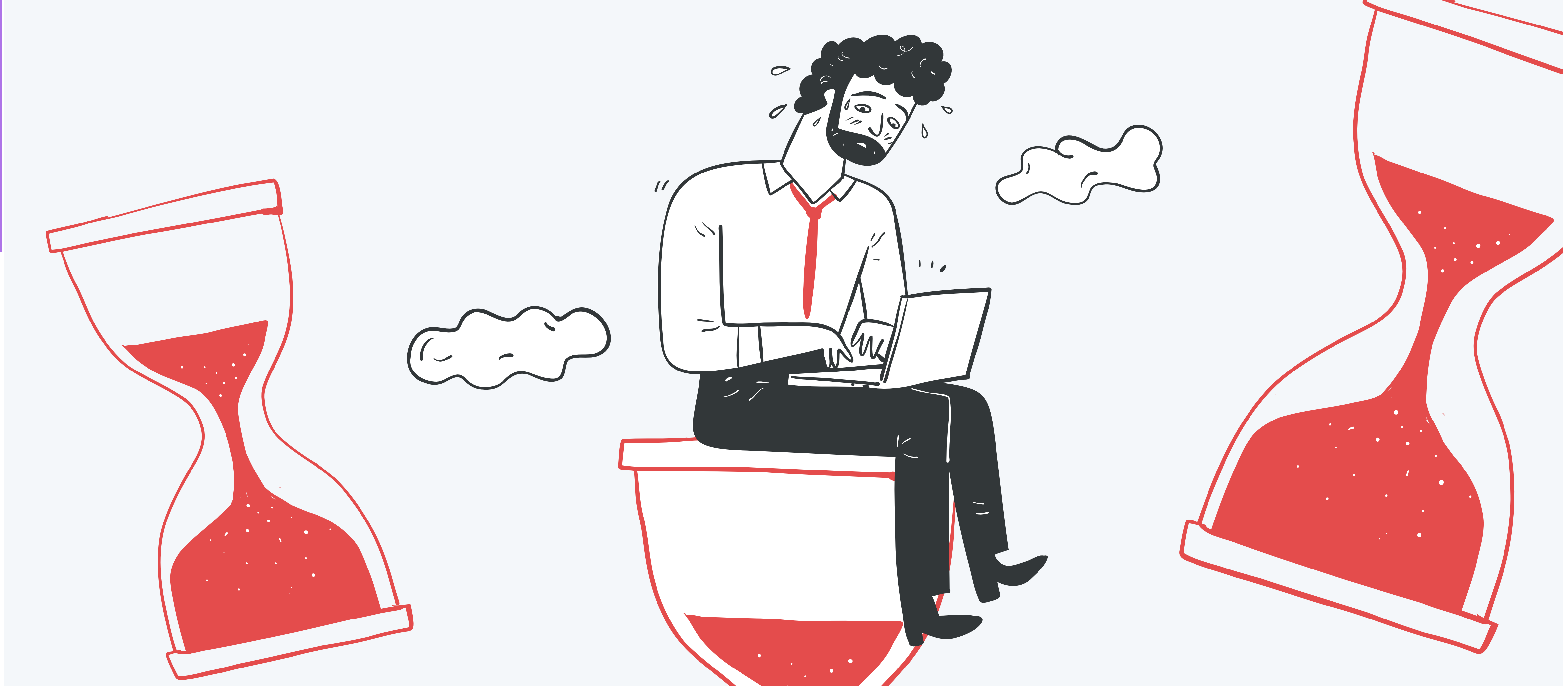
500 lines of code = "looks fine."

Code reviews.

10:58 AM · Nov 5, 2013

7,736 Reposts 171 Quotes 6,518 Likes 66 Bookmarks

The single biggest problem in communication is the illusion that it has taken place.
— George Bernard Shaw



10 😬 Last-Minute Code Changes

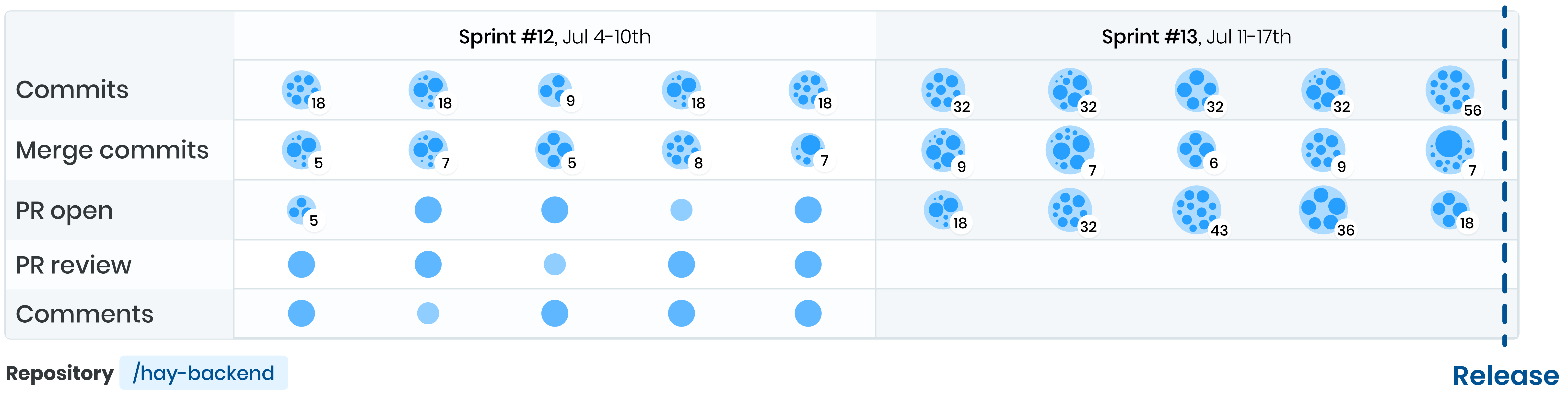
This pattern occurs when a team submits Pull Requests or makes code changes right before a deadline. A rush of activity follows that can disrupt the planned workflow and affect project quality. The source of this problem is that some stakeholders are not interested in the project until the last minute. A week or two before the release, they examine the project and demand changes.

How to recognize it:

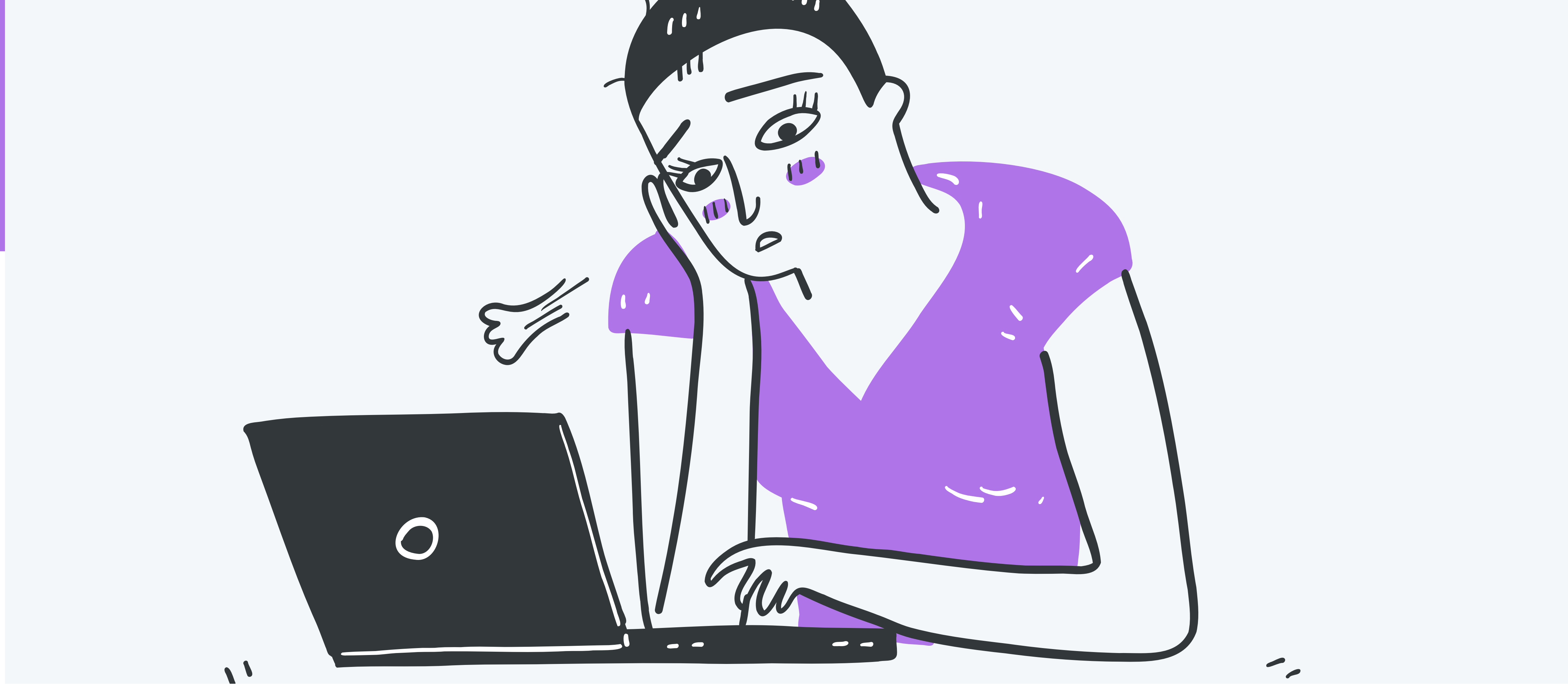
- **Deadline-Driven Commits:** A surge in commits and PRs noticeably close to project deadlines.
- **Stakeholder Demands:** Sudden changes requested by stakeholders as the deadline approaches.
- **Disrupted Workflow:** Observable disruption in the regular workflow pattern prior to release dates.
- **Quality Compromises:** Decrease in code quality or increase in bugs due to rushed changes.
- **Team Stress:** Elevated stress levels and longer working hours as deadlines draw near.
- **Frequent Overhauls:** Major code overhauls or feature additions occurring late in the development cycle.

How to discourage it:

- **Early Stakeholder Engagement:** Involve stakeholders early and regularly throughout the project to prevent late changes.
- **Structured Release Planning:** Implement a well-defined release plan that discourages last-minute alterations.
- **Deadline Buffering:** Incorporate buffer periods before deadlines for unexpected changes and reviews.
- **Regular Progress Updates:** Provide stakeholders with frequent updates to reduce the need for late-stage changes.
- **Team Education:** Educate the team about the impacts of last-minute changes on quality and stress.



Adding last-minute features, whether in response to competitive pressure, as a developer's pet feature, or on the whim of management, causes more bugs in software than almost anything else. — John Robbins



11 🚨 Poor Product Ownership

Poor product ownership is marked by inconsistent guidance from product owners, leading to **frequently shifting project goals and features.** These changes, often introduced mid-development, disrupt workflow and contribute to missed deadlines. This unpredictability can result in scope creep, extending the project's original boundaries and significantly reducing the engineering team's efficiency and productivity.

How to recognize it:

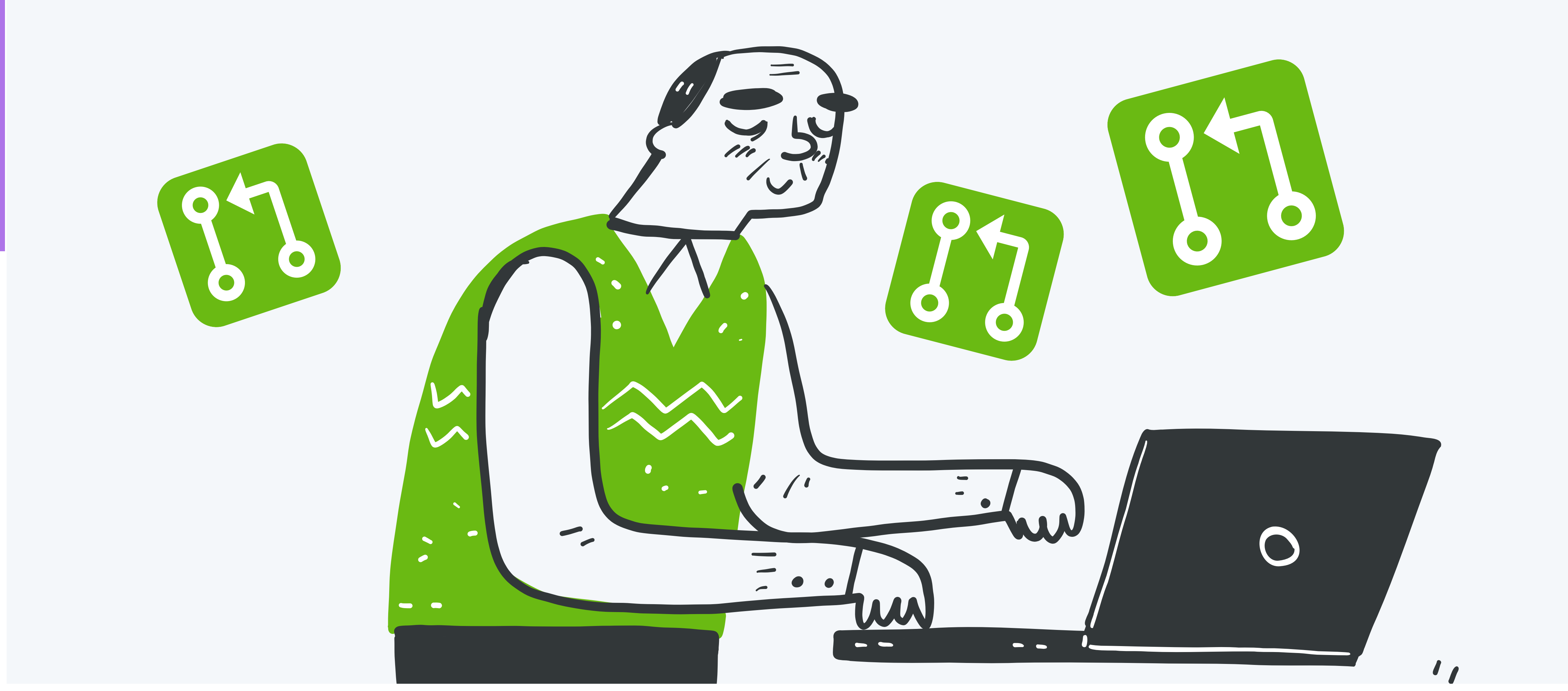
- **Shifting Requirements:** Frequent changes in project requirements even well into the development stages.
- **Unclear Vision:** Lack of a clear, consistent direction from the product owner to the team.
- **Missed Deadlines:** Regular failure to meet deadlines due to changing goals and project scope.
- **Communication Breakdowns:** Inadequate or inconsistent communication between the product owner and development team.
- **Stakeholder Conflicts:** Conflicting demands or feedback from stakeholders not effectively managed by the product owner.
- **Low Team Morale:** Noticeable frustration or confusion within the team due to unclear leadership.
- **Quality Compromise:** Compromised project quality or functionality due to rushed or frequently altered work.

How to discourage it:

- **Define Clear Roles:** Establish and maintain clear roles and responsibilities for product owners.
- **Regular Training:** Provide ongoing training for product owners on effective leadership and communication.
- **Stakeholder Alignment:** Ensure alignment among stakeholders before project initiation and maintain it throughout.
- **Structured Feedback Process:** Create a structured process for collecting and implementing stakeholder feedback.
- **Transparent Communication:** Encourage open and regular communication between product owners and the development team.
- **Set Realistic Goals:** Work with product owners to set achievable goals and realistic timelines.
- **Accountability Measures:** Implement accountability measures for product owners to adhere to the project scope and deadlines.

Don't be silly John...
... it's MY BUDGET.
Sir, it is my job to build the Backlog...
WHAT?!
???
BOSS
PO
DEV
DEV
DEV
SN
DEV
Note to myself: You need a New job...

Overruling the Product Owner?



12 🚨 PRs Open Indefinitely

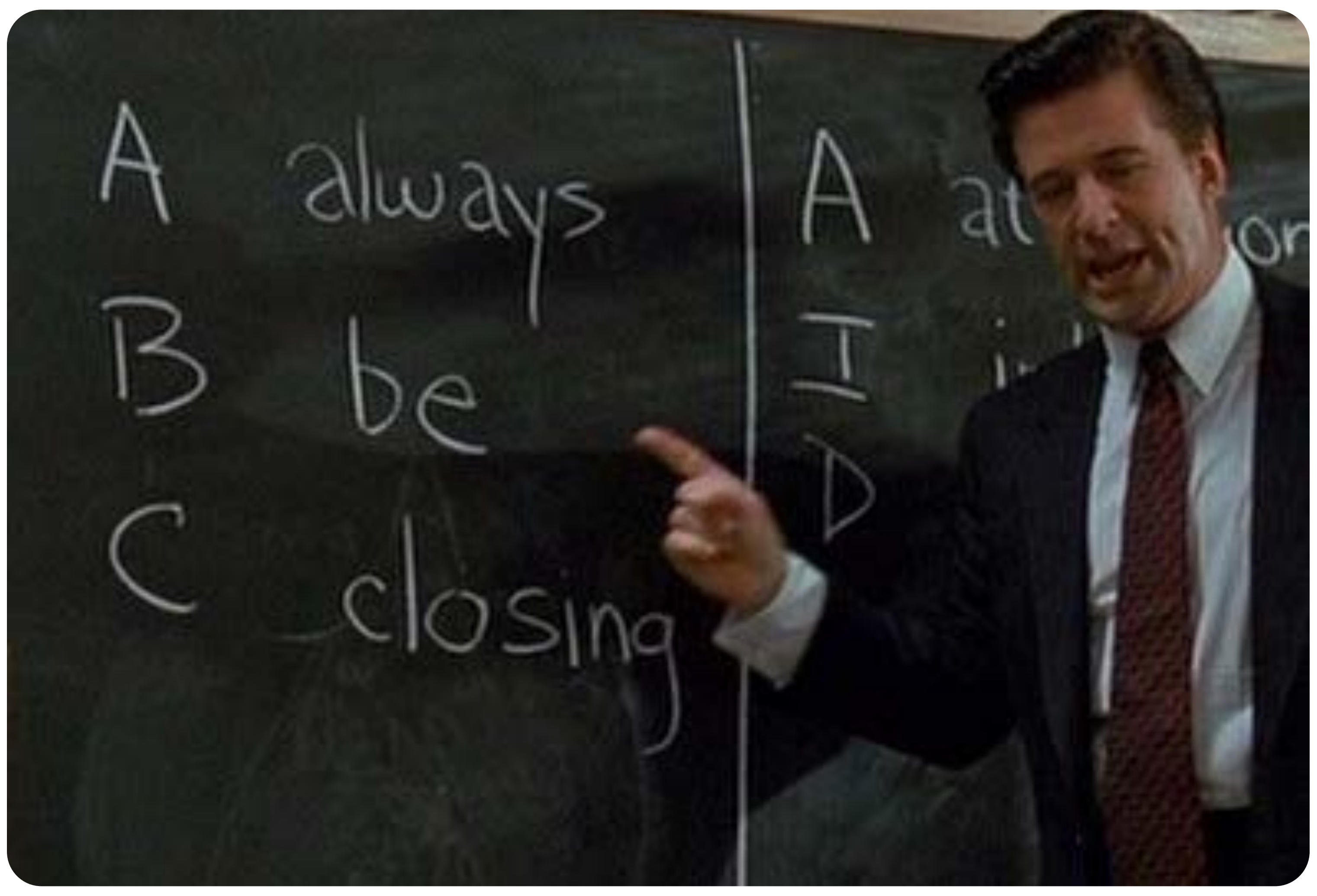
Pull requests that remain open for more than a week often indicate underlying issues in the software development process. This extended duration suggests that the proposed changes in the PR are encountering obstacles, either due to technical challenges, lack of consensus among team members, or indecision. Prolonged open PRs can signify a deeper problem, such as **unclear project guidelines, insufficient communication, or disagreements on implementation strategies.**

How to recognize it:

- **Extended PR Duration:** PRs remaining open well beyond the typical turnaround time for the team.
- **Stalled Discussions:** Lengthy periods of inactivity in the PR discussion thread.
- **Frequent Revisions:** Multiple revisions to the PR without reaching a conclusive resolution.
- **Conflict Indicators:** Signs of disagreement or unresolved issues evident in PR comments.
- **Lack of Decision-Making:** Indecision or lack of clear direction from reviewers or stakeholders.
- **Low Engagement:** Minimal engagement from the team in terms of comments or reviews.
- **Repeated Delays:** Continuous postponement of PR closure without specific reasons.

How to discourage it:

- **Set Clear Deadlines:** Implement firm deadlines for PR review and closure.
- **Regular Review Meetings:** Schedule routine meetings to address and resolve open PRs.
- **Streamline Decision Processes:** Establish a clear decision-making process for accepting or rejecting PRs.
- **Conflict Resolution Strategies:** Develop strategies to efficiently address and resolve conflicts in PRs.
- **Automate Reminders:** Use automated systems to remind reviewers of pending PRs.
- **Encourage Active Participation:** Foster a culture where team members actively participate in PR discussions.
- **Reviewer Accountability:** Hold reviewers accountable for timely feedback and decision-making.



https://www.hanselman.com/blog/always-be-closingpull-requests

13 😬 Low Bus Factor

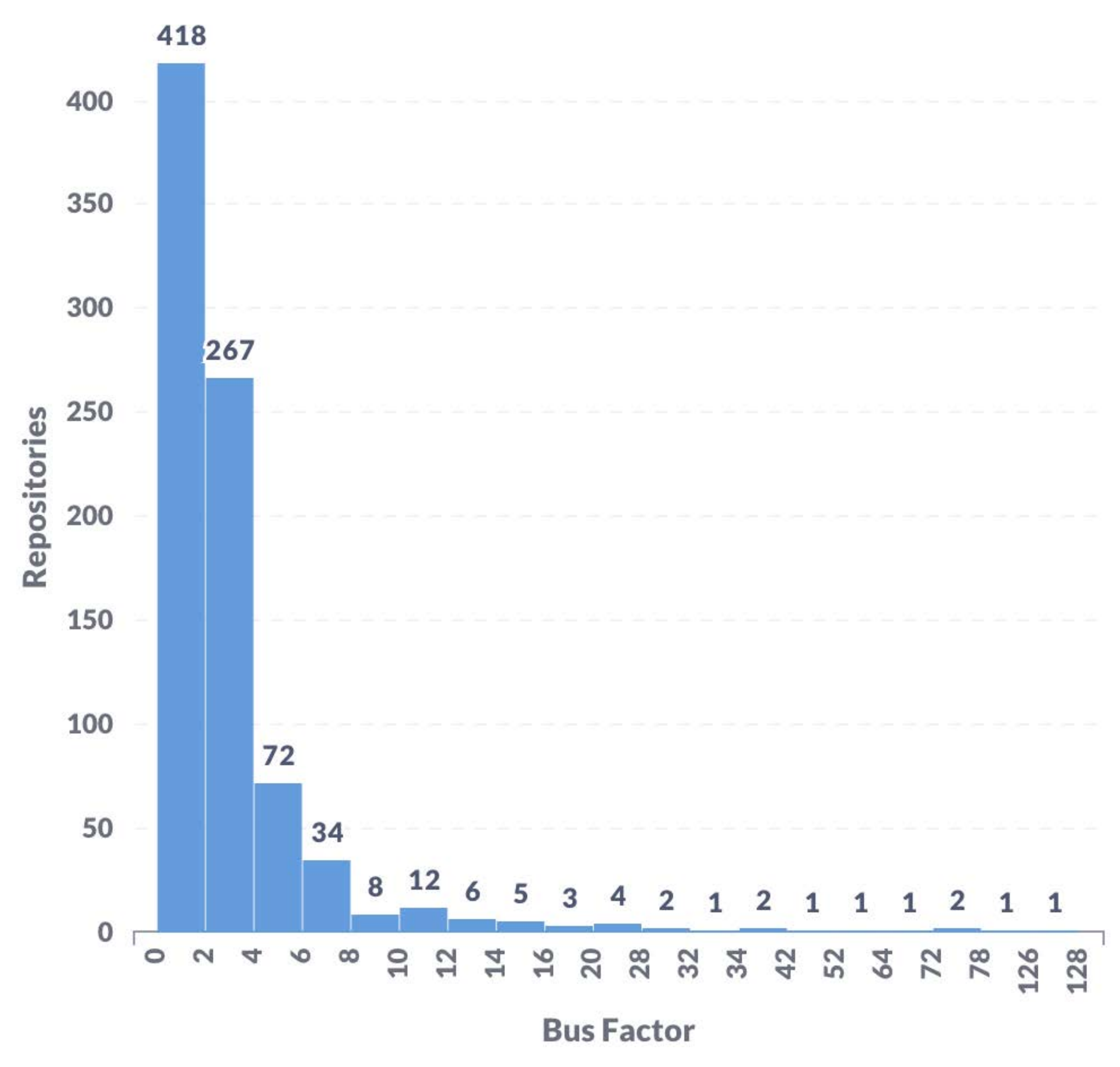
The bus factor is a metric in project management used to gauge the risk associated with **limited knowledge sharing among team members,** indicating the minimum number of individuals whose sudden absence would put the entire project at risk. A low bus factor indicates a risky situation where the departure of one or two key individuals could severely impact the project. A higher bus factor means there's less risk for the project because more people have the knowledge and skills needed to keep things moving, even if someone is absent.

How to recognize it:

- **Key Person Dependency:** Project progress heavily reliant on one or two individuals.
- **Limited Knowledge Sharing:** Lack of widespread knowledge transfer and skill sharing among team members.
- **Narrow Expertise Pools:** Only a few team members are experts in critical areas of the project.
- **Stalled Progress in Absence:** Noticeable slowdown or halt in work when specific individuals are unavailable.
- **Concentrated Task Allocation:** Consistent assignment of crucial tasks to the same few individuals.
- **Uneven Training Levels:** Disparity in skill and knowledge levels across the team.
- **Reluctance to Delegate:** Key individuals often resist delegating important tasks to others.

How to discourage it:

- **Promote Cross-Training:** Encourage team members to learn different aspects of the project.
- **Implement Pair Programming:** Pair programming sessions for knowledge sharing and skill development.
- **Document Processes Thoroughly:** Ensure documentation of processes and project knowledge.
- **Rotate Roles Regularly:** Periodically rotate team members through different roles and tasks.
- **Foster a Collaborative Culture:** Create an environment where knowledge sharing is expected and rewarded.
- **Regular Knowledge Sharing Sessions:** Schedule sessions for team members to share insights and learnings.

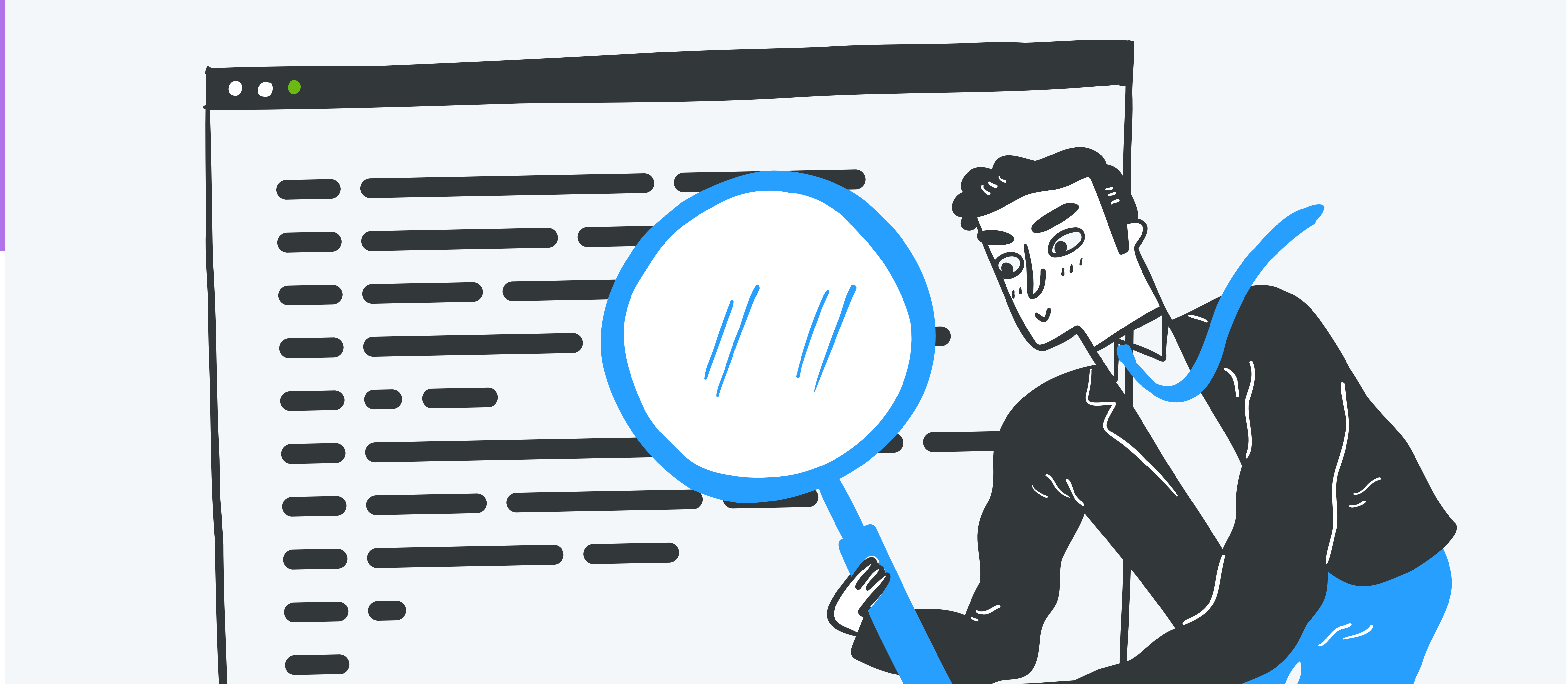


Bus factor for top 1000 GitHub open source projects

https://www.metabase.com/blog/bus-factor

"Everyone has transferable commodity knowledge. Sharing your unique expertise and making introductions for someone creates a lasting legacy."
— Marsha Blackburn

14

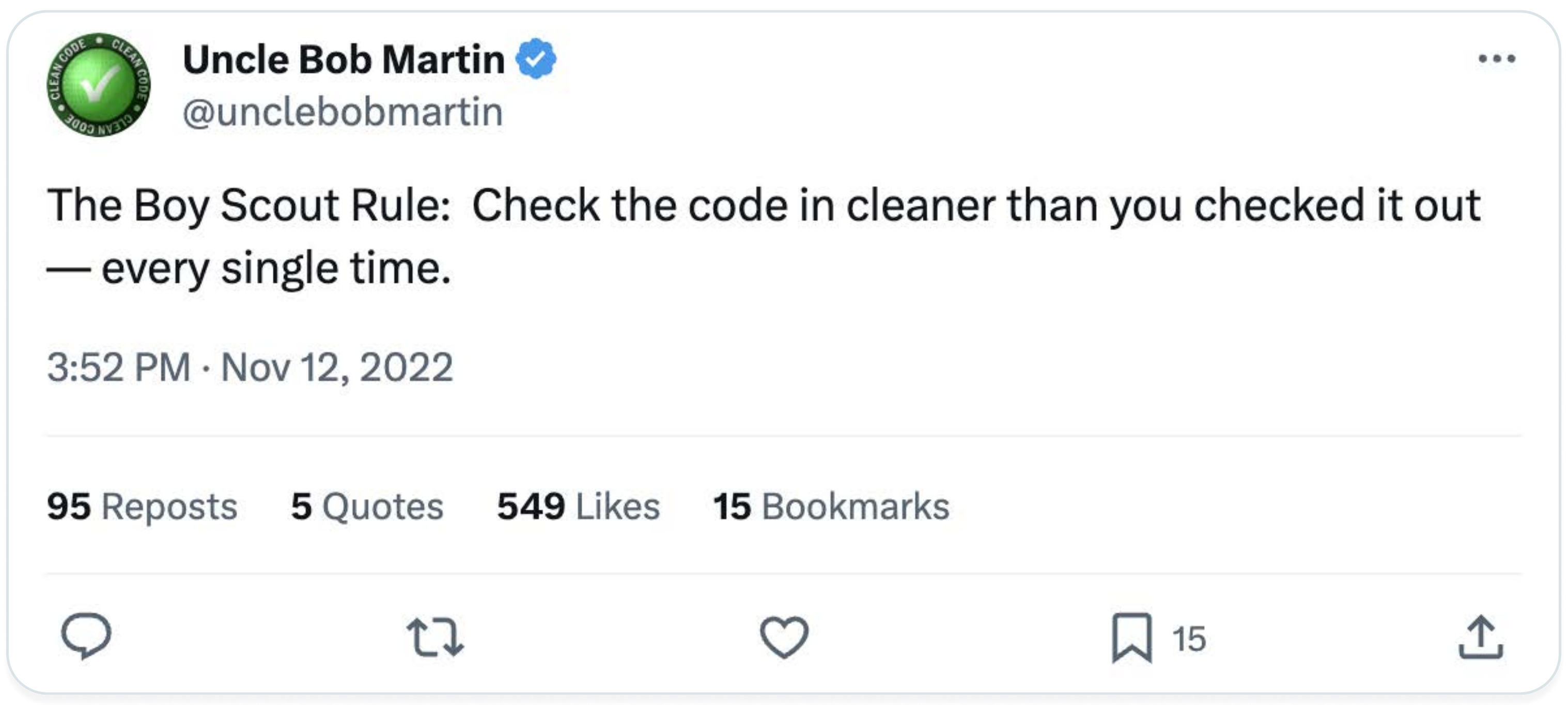


🍀 Harmonizing Code Maintenance with New Code Development

Developers often dedicate some of their time to cleaning code in order to enhance the codebase. However, stakeholders aren't concerned with "cleaner code" or "cleaning up the codebase" — they want to see new features developed.

How to recognize it:

- **Balanced Commit Patterns:** Regular commits showing a mix of new features and code maintenance tasks.
- **Clear Refactoring Logs:** Documentation or commit messages clearly indicating refactoring or code cleanup activities.
- **Stakeholder Feedback Integration:** Visible incorporation of stakeholder feedback focused on new features and progress.
- **Code Quality Metrics:** Consistent improvement in code quality metrics alongside the development of new features.
- **Team Discussions:** Frequent team discussions balancing new feature development with codebase improvements.
- **Time Allocation:** Set portions of development time dedicated specifically to code maintenance and refactoring.



Uncle Bob Martin
@unclebobmartin

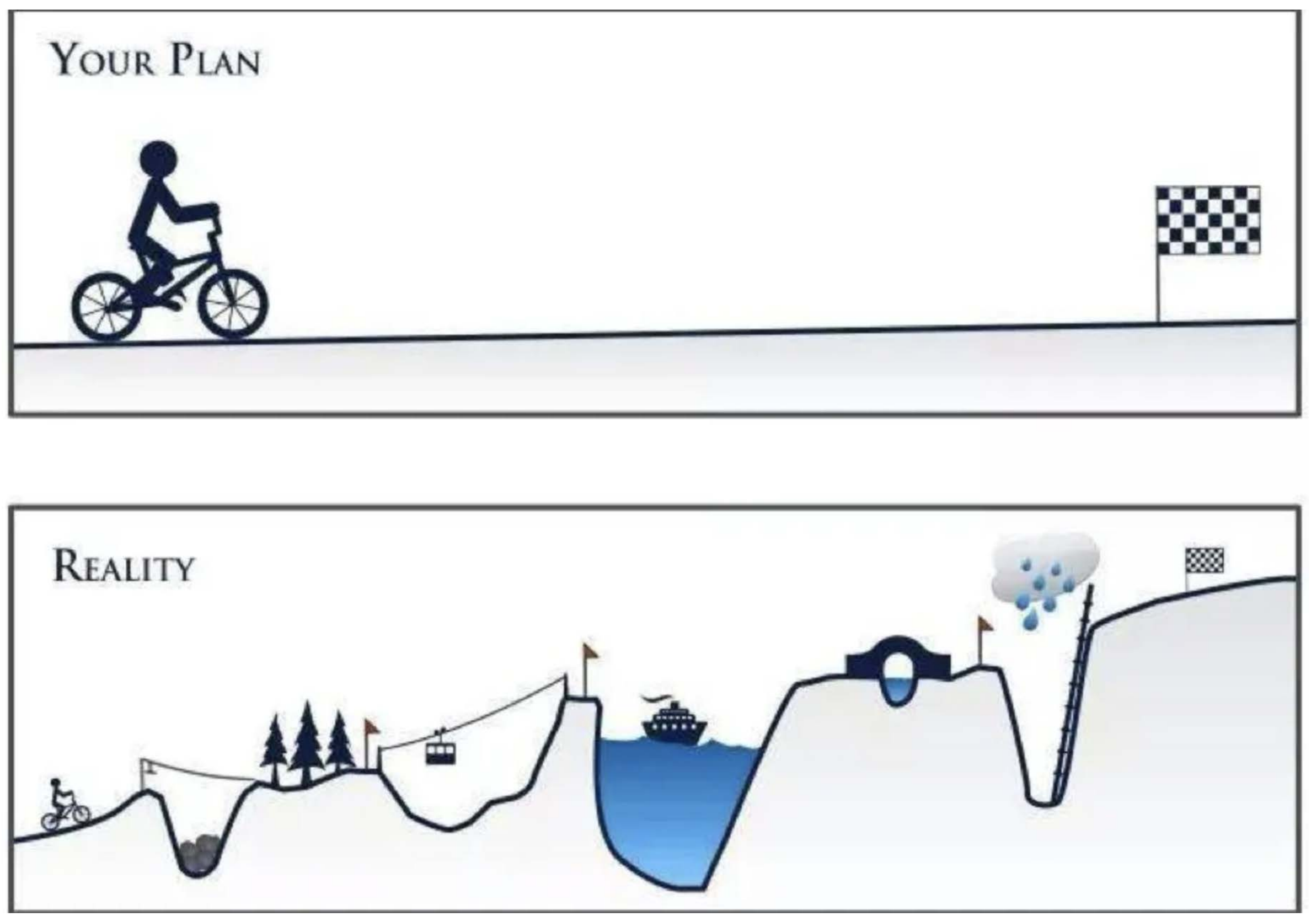
The Boy Scout Rule: Check the code in cleaner than you checked it out — every single time.

3:52 PM · Nov 12, 2022

95 Reposts 5 Quotes 549 Likes 15 Bookmarks

How to encourage it:

- **Set Priorities:** Communicate the importance of new feature development and code maintenance.
- **Allocate Time:** Dedicate time slots for code maintenance along with feature development.
- **Educate Stakeholders:** Inform stakeholders about code maintenance's long-term benefits.
- **Celebrate Milestones:** Reward efforts to improve the codebase.
- **Provide Tools:** Equip the team with efficient code refactoring and analysis tools.
- **Encourage Collaboration:** Foster a team environment for joint code improvements.



https://medium.com/@michalrychlik/new-features-vs-maintenance-developers-perspective-b6ea110c58b9

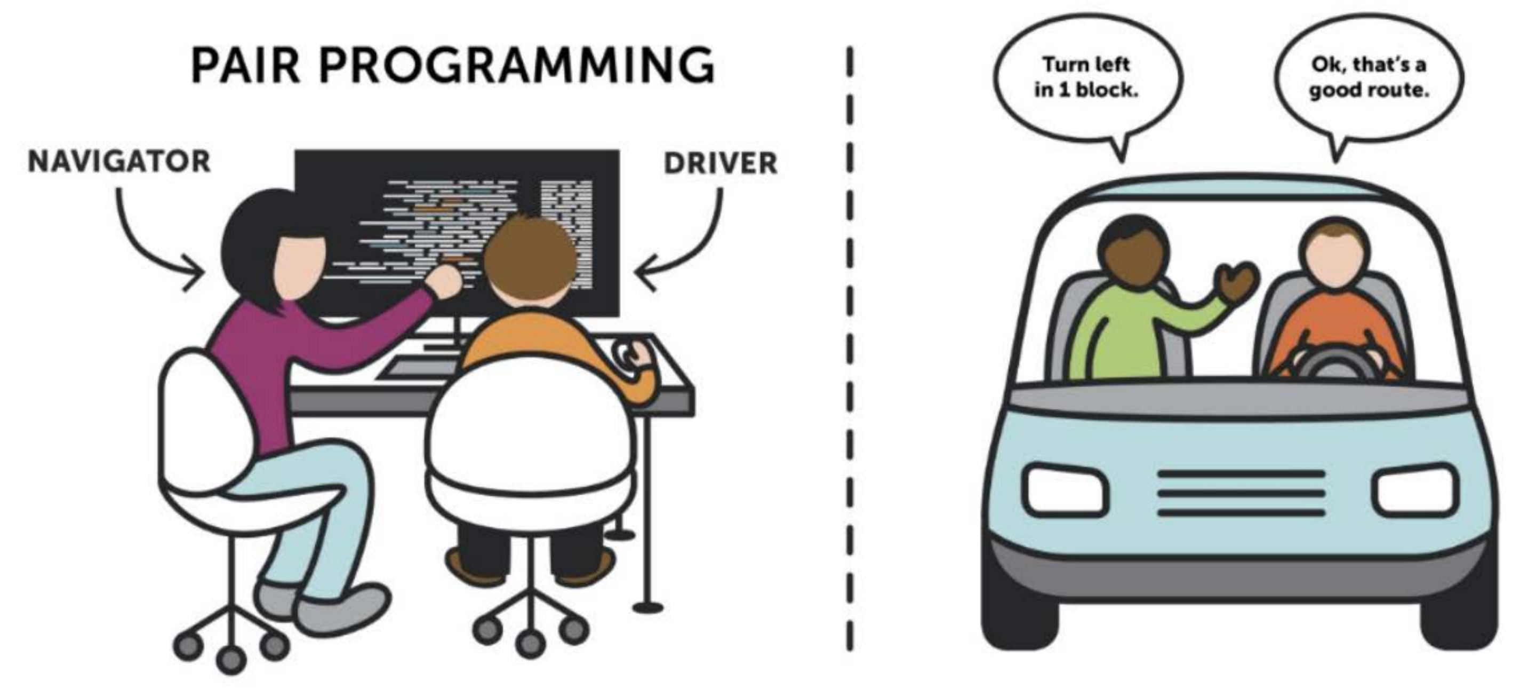
15

🍀 Pair Programming

Pair programming is a collaborative coding practice where two developers work together at one workstation. One, the "driver," writes code while the other, the "observer" or "navigator," reviews each line of code as it is typed in. The roles are frequently switched during a session. This approach not only **enhances code quality through real-time review** but also **facilitates knowledge sharing** and reduces the likelihood of overlooked mistakes.

How to recognize it:

- **Two Developers, One Workstation:** Regularly seeing two team members collaborating at a single computer.
- **Active Role Switching:** Observing team members frequently exchange roles of typing and reviewing.
- **Joint Problem Solving:** Noticing pairs discussing and solving coding challenges together.
- **Knowledge Transfer:** Seeing less experienced developers rapidly improving with the aid of a partner.
- **High Engagement Levels:** High levels of engagement and interaction between two team members.
- **Reduced Code Errors:** Fewer mistakes and bugs in code segments developed through pair programming.
- **Balanced Contribution:** Both members of the pair contributing ideas and solutions equally.



https://unruly.co/blog/article/2019/08/27/what-is-pair-programming/

How to encourage it:

- **Scheduled Pairing Sessions:** Allocate specific times for developers to engage in pair programming.
- **Diverse Pairing Combinations:** Rotate pairing partners to spread knowledge and experience across the team.
- **Integration in Agile Practices:** Incorporate pair programming in sprints and Agile methodologies.
- **Promote Benefits:** Educate the team on the advantages of pair programming for learning and quality.
- **Recognition and Rewards:** Acknowledge and reward successful outcomes achieved through pair programming.
- **Pairing for Complex Tasks:** Assign challenging tasks to be tackled via pair programming.
- **Provide Necessary Resources:** Ensure teams have the tools and environment conducive to effective pairing.

Pair programming is a dialogue between two people trying to simultaneously program (and analyze and design and test) and understand together how to program better. It is a conversation at many levels, assisted by and focused on a computer.
— Kent Beck

Leadership Behaviors

16 🚨 Misperceive Project Status

There is nothing worse than an engineering leader thinking that a project is on track for success, especially without the confirmation of data. Yet on delivery day, there are unresolved core issues. In many cases, this leadership behavior appears when leaders are too optimistic about a project without really finding out the project's status.

How to recognize it:

- **Over-Optimistic Projections:** Leader consistently expresses confidence in project success without supporting data.
- **Lack of Detailed Inquiries:** Rarely asks for in-depth updates or specific details about project progress.
- **Dismissal of Concerns:** Tends to overlook or downplay issues raised by team members.
- **Infrequent Team Consultations:** Engages with the team less often to understand the real status of the project.
- **Reliance on Surface Metrics:** Focuses more on superficial success indicators than on substantive project details.
- **Surprise at Delivery Issues:** Often caught off guard by problems that arise during final project stages.
- **Gap in Expectation vs. Reality:** A noticeable discrepancy between the leader's perception and actual project outcomes.

https://www.pm-partners.com.au/how-to-avoid-project-failure-by-heeding-the-warning-signs/

How to discourage it:

- **Data-Driven Decision Making:** Encourage leaders to base their project assessments on concrete data and metrics.
- **Regular Detailed Updates:** Implement a system for regular, detailed project status reports and reviews.
- **Foster Open Communication:** Create an environment where team members can openly share challenges and progress.
- **Leadership Training:** Provide training on effective project oversight, emphasizing the importance of realistic assessments.
- **Balanced Optimism:** Teach leaders to balance optimism with a realistic appraisal of project challenges.
- **Stakeholder Feedback:** Regularly include stakeholder feedback in project assessments to provide a fuller picture.
- **Reward Accurate Assessment:** Recognize and reward leaders who accurately assess and communicate project status.

"The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge."
— Stephen Hawking

17

🍀 Mentorship

Mentorship is invaluable but challenging to sustain amidst competing project demands and deadlines. In software houses, traditional mentorship might be more viable, whereas in project-based environments, pair programming is often more practical. In any case, effective mentors are essential to balance these approaches and ensure ongoing personal and professional development within the team.

How to recognize it:

- **Active Guidance:** Leaders regularly engage in one-on-one sessions for personal and career guidance.
- **Knowledge Sharing:** Frequent instances of leaders imparting technical and industry knowledge to team members.
- **Supportive Feedback:** Providing constructive feedback that fosters growth and development in mentees.
- **Pairing Seniors with Juniors:** Strategic pairing of experienced developers with juniors for hands-on learning.
- **Career Development Focus:** Leaders actively involved in discussing and planning team members' career paths.
- **Recognition of Progress:** Acknowledging and celebrating the professional growth and achievements of team members.
- **Open-Door Policy:** Leaders maintain an approachable demeanor, encouraging team members to seek advice.

How to encourage it:

- **Mentorship Programs:** Establish structured mentorship programs that match mentors with mentees based on skills and career goals.
- **Training for Mentors:** Provide training to potential mentors to equip them with effective mentoring techniques.
- **Incentivize Mentoring:** Recognize and reward mentors for their contributions to the growth of their mentees.
- **Create Mentoring Opportunities:** Allocate time for mentoring activities within the regular work schedule.
- **Pair Programming Sessions:** Encourage pair programming, especially for complex tasks, as a form of practical mentorship.
- **Feedback Culture:** Foster a culture where continuous feedback is part of the team's routine.
- **Leadership Support:** Ensure that top leadership endorses and actively supports the mentorship initiative.

“*Our chief want in life is somebody who will make us do what we can.*”
— Ralph Waldo Emerson

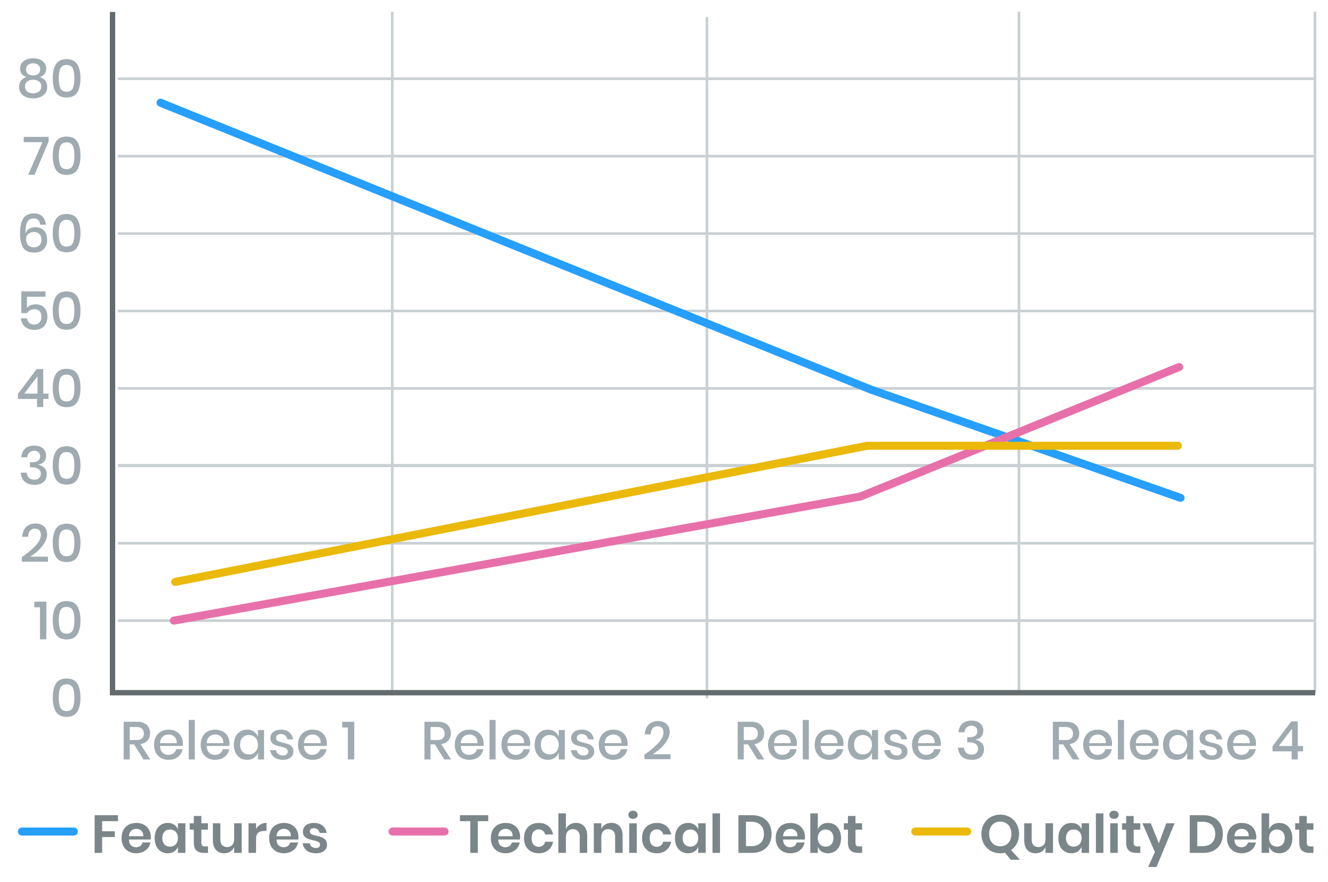
18

😬 Prioritizing Feature Delivery Over Code Longevity

Developers frequently face pressure to deliver features quickly, which can lead to **neglecting the long-term reliability and quality of the code.** Failure to future-proof code can result in functionality problems, leading to unclear code, maintenance challenges, and a lack of standardization.

How to recognize it:

- **Rushed Releases:** Frequent pushing of features to production with minimal testing and refinement.
- **Technical Debt Accumulation:** Increasing instances of code-related problems due to rushed or incomplete development.
- **Overemphasis on Deadlines:** Leadership focuses more on meeting deadlines than on the quality of the deliverables.
- **Neglect of Code Maintenance:** Little to no time allocated for code maintenance and refactoring tasks.
- **Feedback Ignored:** Developers' concerns about code quality and long-term viability are often overlooked.
- **Short-Term Focus in Planning:** Project plans prioritize immediate feature completion over sustainable development practices.



https://www.researchgate.net/profile/Ken-Power/publication/271437496/figure/fig2/AS:731750229630976@1551474258349/Mounting-technical-debt-impacts-feature-velocity-over-time.png

How to discourage it:

- **Balance Speed with Quality:** Emphasize the importance of balancing quick feature delivery with code quality.
- **Allocate Time for Refactoring:** Schedule dedicated time for code maintenance and refactoring in project timelines.
- **Educate on Long-Term Impacts:** Highlight the long-term consequences of neglecting code quality on project sustainability.
- **Implement Quality Metrics:** Introduce and track metrics that measure code quality and long-term viability.
- **Encourage Regular Code Reviews:** Foster a culture where thorough code reviews are a standard part of the development process.
- **Recognize Quality Contributions:** Acknowledge and reward efforts that enhance code quality and longevity.

Shortcuts make long delays
— J.R.R. Tolkien

19

🚨 Onboarding Too Much Information, Too Quickly

New teammates added to a project or who recently joined the company often face a steep learning curve. A common pitfall for leaders is when they onboard in a way that **prioritizes volume of information transfer rather than active learning** or a learn-as-you-go strategy. This can significantly prolong the onboarding process and delay the time-to-productivity, which can sometimes be 8-12 months.

How to recognize it:

- **Overwhelming Initial Sessions:** New hires are bombarded with extensive information in their first few days.
- **Delayed Productivity:** Noticeable delay in new team members becoming productive contributors to projects.
- **High Information Retention Expectations:** Expecting new hires to retain and apply large volumes of information rapidly.
- **Lack of Practical Engagement:** Limited opportunities for hands-on learning or involvement in real project tasks.
- **New Hires' Stress Levels:** Visible stress or confusion among new hires due to information overload.
- **Little Time for Reflection:** Minimal downtime allocated for new team members to process and understand information.
- **Feedback Ignored:** Disregarding new hires' feedback about the pace and content of onboarding.

Tell me and I forget, teach me and I may remember, involve me and I learn.
— Benjamin Franklin

How to discourage it:

- **Structured Phased Approach:** Implement a phased onboarding process that gradually increases complexity.
- **Focus on Active Learning:** Prioritize hands-on tasks and real-world project involvement early on.
- **Regular Check-Ins:** Schedule frequent check-ins to assess understanding and adjust the onboarding pace.
- **Buddy System:** Pair new hires with experienced team members for guided learning and support.
- **Feedback-Driven Adjustments:** Regularly solicit and act on feedback to improve the onboarding experience.
- **Set Realistic Expectations:** Clearly communicate that learning and productivity growth will be gradual.
- **Encourage Questions and Exploration:** Create a welcoming environment for new hires to ask questions and explore.

Recommended Additional Reading:

"Inside Google's New Hire Onboarding Process"

"Mastering the Art of Onboarding: Strategies for Software Engineers"

20

😬 Reckless Deployment

Reckless deployment often occurs when the urgency to deliver quick results compromises code quality. This mindset encourages teams to **accelerate the deployment process, often bypassing critical checks** and balances, which can compromise the integrity and reliability of the final product. This approach not only risks introducing bugs and system failures but also undermines the long-term sustainability of the software.

How to recognize it:

- **Frequent Hotfixes:** Rapid deployment followed by numerous urgent fixes.
- **Code Quality Compromises:** Code is deployed with known issues to meet tight deadlines.
- **High Bug Rates Post-Deployment:** Increase in user-reported issues soon after new releases.
- **Shortened or Skipped Testing:** Testing phases are rushed or overlooked before deployment.
- **Pushback Against Quality Control:** Resistance to implementing thorough quality assurance processes.
- **Deployment Over Reflection:** Quick deployments without adequate review of feedback or performance metrics.
- **User Feedback Neglect:** User or stakeholder feedback on issues is often ignored or minimized.

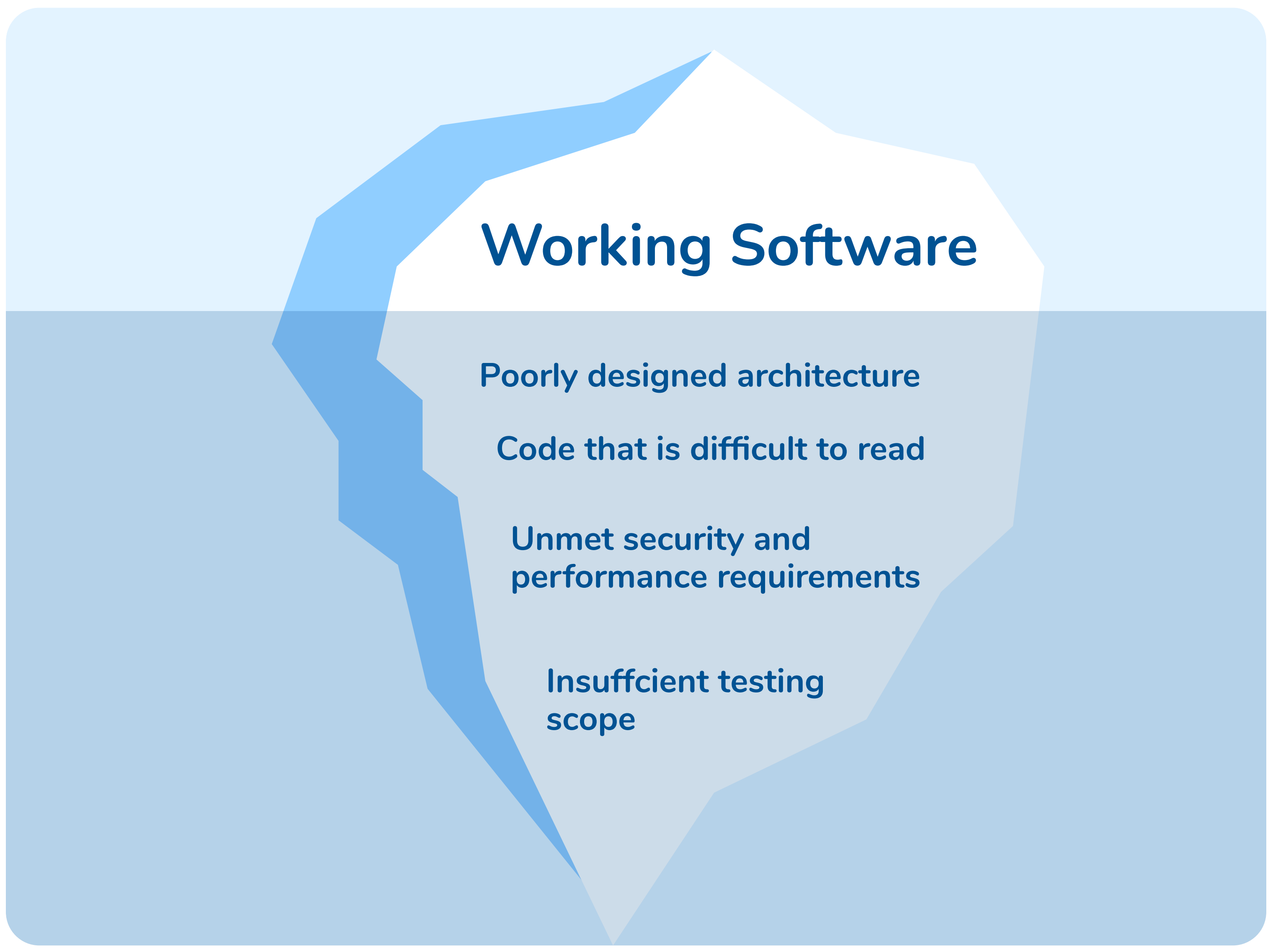
How to discourage it:

- **Emphasize Quality in Culture:** Cultivate a team culture where quality is valued as much as speed.
- **Implement Rigorous Testing Protocols:** Establish and enforce comprehensive testing before deployment.
- **Set Realistic Timelines:** Ensure project timelines are realistic, allowing for thorough development and testing.
- **Educate on Long-Term Impacts:** Highlight the long-term costs and risks associated with reckless deployment.
- **Reward Diligence:** Recognize and reward teams that prioritize code integrity and reliability.
- **Continuous Improvement Focus:** Encourage a mindset of continuous improvement.
- **Stakeholder Engagement:** Involve stakeholders in understanding the importance of quality in deployment processes.

Even the deployment process should be automated and carefully designed to achieve high availability

Availability Level	Downtime per Year
90%	~ 36.5 days
99%	~ 3.65 days
99.9%	~ 8.76 hours
99.99%	~ 52.56 minutes
99.999% (“5-Nines”)	~ 5 m and 26 s

https://blog.hireplicity.com/blog/technical-debt-software-development


Working Software
Poorly designed architecture
Code that is difficult to read
Unmet security and performance requirements
Insuffcient testing scope

Even the deployment process should be automated and carefully designed to achieve high availability

https://assets-global.website-files.com/643d1b14f7e2ef6308449a50/6489dbe1a545a2eeb7c89e95_Cover%20Blogs.jpg

21 🍀 Learn to Say "No"

Engineering leaders, PMs, or POs have to learn how to say "No" to unrealistic client feature requests. Reasons for this include to protect project integrity, conserve resources, and ensure technical soundness. Rejecting unfeasible ideas or those misaligned with the product's vision is crucial to **avoid wasting time and compromising quality.**

Saying "No" starts with:

- **Promoting Open Communication:** Encourage an environment for open dialogue to assess feature request feasibility and impacts.
- **Implementing Decision-Making Frameworks:** Provide teams with standardized tools for evaluating feature requests against strategic alignment and resource constraints.
- **Suggesting Constructive Alternatives:** Train leaders to offer feasible alternatives, turning rejections into collaborative problem-solving opportunities.
- **Prioritization of Resources:** Allocate resources based on strategic priorities, steering clear of decisions driven solely by client pressure.
- **Realistic Commitments:** Ensure leadership commits realistically, focusing on achievable deliverables instead of over-promising.

How to encourage it:

- **Empowerment and Support:** Empower leaders to make decisions aligned with project goals, offering support when they refuse unreasonable requests.
- **Training in Negotiation:** Provide training in negotiation and conflict resolution to handle client interactions effectively.
- **Emphasize Strategic Goals:** Regularly reinforce the importance of adhering to strategic goals and product vision.
- **Client Education:** Educate clients about the product development process to set realistic expectations.
- **Recognition for Tough Decisions:** Acknowledge and praise leaders when they make tough but necessary decisions.

It's only by saying 'no' that you can concentrate on the things that are really important.
— Steve Jobs

Saying NO as a Product Owner

22 🍀 Continuous Learning

In reality, the rush to meet deadlines often sidelines learning, yet teams that embrace Investment Time or other learning initiatives tend to deliver more innovative and robust solutions more rapidly. Leaders who cultivate a positive learning environment play their part in closing knowledge gaps. They also create a workplace that's open, truthful, and psychologically secure.

How to recognize continuous learning opportunities:

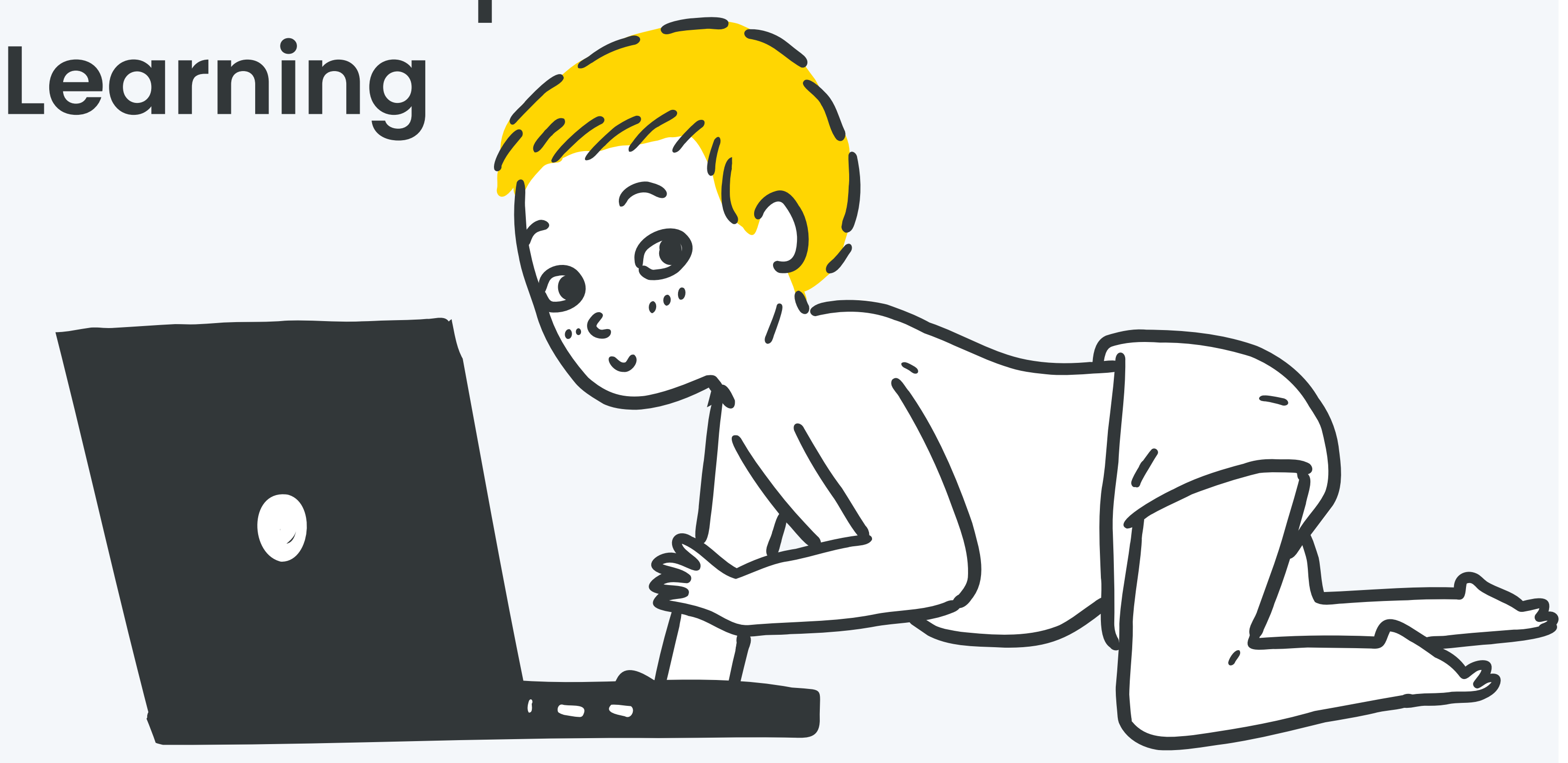
- **Regular Skill Assessments:** Conduct assessments to identify skills gaps and target areas for growth.
- **Industry Trends Awareness:** Stay updated on the latest industry developments and technologies for potential learning.
- **Feedback Loops:** Utilize feedback from team members to identify learning needs and opportunities.
- **Performance Metrics:** Analyze performance data to pinpoint areas where additional training could enhance productivity.
- **Collaboration and Peer Learning:** Recognize moments in teamwork that could evolve into learning opportunities.
- **Employee Interests:** Pay attention to the interests and inclinations of team members for tailored learning experiences.
- **Project Challenges:** Identify complex project aspects that could benefit from focused learning or upskilling.

How to encourage it:

- **Create Learning Plans:** Develop and implement individual and team learning plans aligned with business goals.
- **Allocate Learning Time:** Set aside dedicated time for team members to engage in learning activities.
- **Provide Resources:** Offer access to courses, workshops, and other educational materials.
- **Mentorship Programs:** Establish mentorship arrangements that facilitate knowledge sharing and personal development.
- **Encourage Experimentation:** Create a safe space for trying new methods and technologies as part of learning.
- **Recognize and Reward Learning:** Acknowledge progress in learning and reward efforts that contribute to personal and team growth.
- **Lead by Example:** Actively engage in learning yourself to demonstrate its value and importance.

An investment in knowledge always pays the best interest.
— Benjamin Franklin

Great Developers Never Stop Learning



23

🍀 Don't Hire "Ninja" Developers

"Ninja" developers are often characterized as highly skilled and productive coders who work quickly and with a high degree of autonomy. They are typically seen as experts in their field, capable of tackling complex coding challenges with ease. However, the term can also imply a **lone-wolf mentality,** where such developers **work in isolation, prioritize their own coding methods**, never attend meetings, or often disregard team dynamics or collaborative processes.

Characteristics of "ninja" developers:

- **Exceptional Coding Skills:** Possess high-level coding abilities and a strong capability to solve complex problems.
- **High Autonomy:** Work independently with minimal guidance or supervision.
- **Lone-Wolf Mentality:** Often operate in isolation, preferring to tackle challenges alone.
- **Methodical Approach:** Have distinct coding methods and practices, often unique to them.
- **Meeting Avoidance:** Typically avoid or minimally participate in team meetings and collaborative sessions.
- **Individualistic Mindset:** Focus on personal coding achievements over team goals.
- **Limited Collaboration:** Show reluctance or disinterest in engaging with team dynamics or joint problem-solving.

Reasons not to hire "ninja" developers:

- **Team Dynamic Disruption:** Can disrupt team cohesion and collaborative workflows.
- **Knowledge Siloing:** Their solo work style risks creating knowledge silos within teams.
- **Communication Gaps:** Often create communication challenges due to their isolated working style.
- **Inflexibility:** May resist adapting to team methodologies or contributing to shared codebases.
- **Overdependence Risk:** Can lead to over-reliance on a single individual for critical tasks.
- **Reduced Team Learning:** Their autonomous nature can limit knowledge sharing and collective team growth.
- **Inconsistent with Agile Principles:** Their working style often conflicts with Agile methodologies focused on teamwork and adaptability.

Talent wins games, but teamwork and intelligence win championships.
— Michael Jordan

24
🚨 When Helping Harms Development

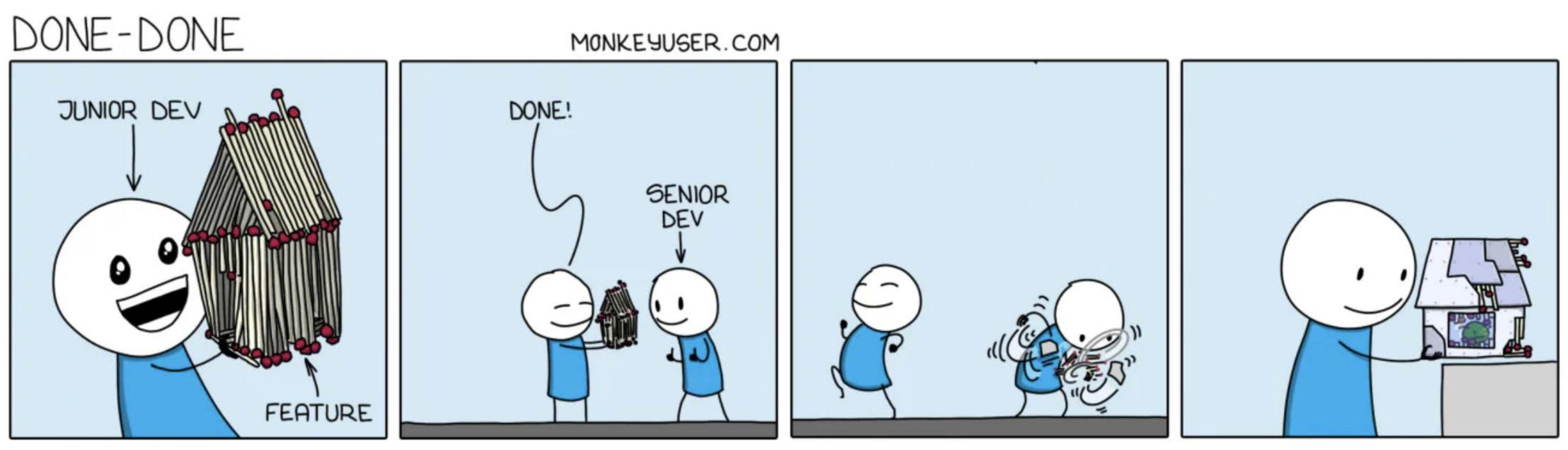
Managers, in their efforts to assist, sometimes inadvertently hinder another developer's progress. This can happen when they assume all the challenging responsibilities, including providing excessive help, but doesn't delegate some duties to less experienced colleagues. The excessive time spent by the manager isn't the concern, as they usually are adept at time management; the real problem lies in the limited opportunities for less experienced individuals to acquire experience.

How to recognize it:

- **Over-Engagement by Seniors:** Senior staff frequently intervene in tasks typically suited for junior developers.
- **Limited Challenges for Juniors:** Less experienced developers receive few challenging assignments, impeding growth.
- **Reduced Problem-Solving Opportunities:** Juniors seldom get chances to solve complex problems independently.
- **Senior Task Monopoly:** Critical and complex tasks are consistently handled by more experienced individuals.
- **Lack of Delegation:** Seniors exhibit reluctance to delegate significant responsibilities to junior team members.
- **Minimal Junior Progress:** Slow professional development and skill advancement among junior developers.
- **Uneven Workload Distribution:** Disproportionate distribution of challenging tasks favoring senior developers.

How to discourage it:

- **Encourage Controlled Autonomy:** Allow junior developers to handle tasks independently within defined parameters.
- **Implement Mentorship Programs:** Pair juniors with seniors for guidance, not takeover.
- **Monitor Task Allocation:** Ensure an equitable distribution of challenging tasks across all experience levels.
- **Promote a Learning Culture:** Create a culture that values learning through doing, even if it means making mistakes.
- **Set Specific Growth Goals:** Define clear development and learning objectives for junior team members.
- **Foster Problem-Solving Skills:** Encourage juniors to find solutions before seeking help, fostering critical thinking.
- **Acknowledge and Reward Progress:** Recognize and reward junior developers for successfully tackling complex tasks.



"Give a man a fish, and you feed him for a day. Teach a man to fish, and you feed him for a lifetime."

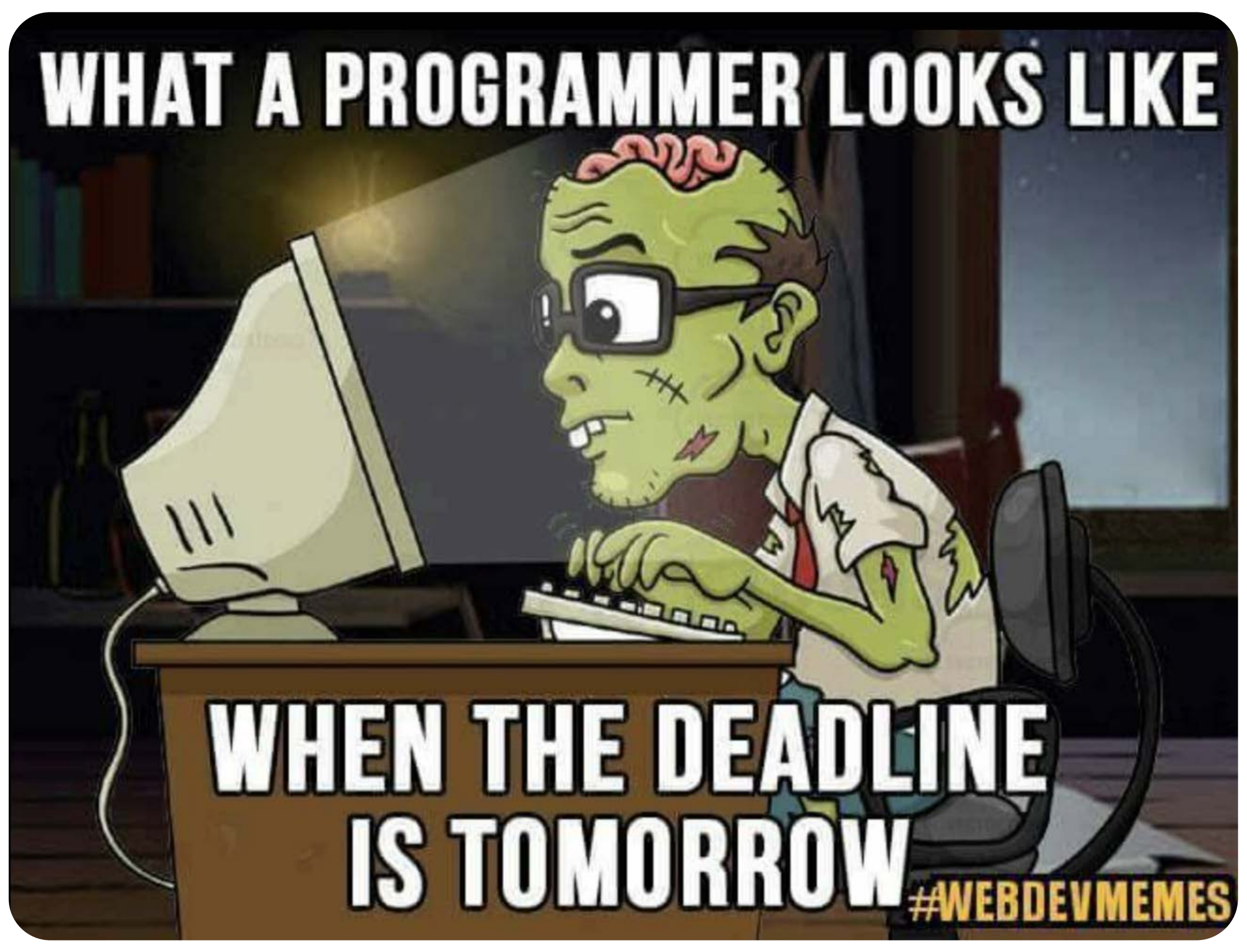
25

🚨 Poor Capacity Allocation During Crunch Time

In periods of intense production, leaders often face the challenging task of managing an increased workload alongside limited resources. However, when this task is not handled effectively, it can lead to a situation where the **team's workload becomes unmanageable**. As a result, multiple tasks and projects are deemed critical at the same time, putting undue pressure on the development team.

How to recognize it:

- **Simultaneous Deadlines:** Multiple critical tasks assigned with the same urgent deadlines.
- **Increased Team Stress:** Noticeable rise in stress levels and fatigue among team members.
- **Unbalanced Workloads:** Some developers are overwhelmed while others are underutilized.
- **Missed Deadlines:** Frequent inability to meet project deadlines due to overcapacity.
- **Quality Compromise:** Rushed work leading to a decline in code quality and increased errors.
- **Frequent Overtime:** Regular need for extended work hours or working weekends.
- **Reduced Collaboration:** Diminished team collaboration due to excessive individual workloads.



https://www.facebook.com/CodeChef/photos/a.10150302285647799/10155368459827799/?type=3

How to discourage it:

- **Prioritize Tasks Effectively:** Clearly identify and communicate task priorities to avoid simultaneous critical deadlines.
- **Balance Workloads:** Regularly assess and distribute workloads evenly among team members.
- **Plan for Peaks:** Anticipate peak times and plan resource allocation in advance.
- **Empower Delegation:** Encourage team leads to delegate tasks appropriately to manage workloads.
- **Monitor Stress Levels:** Keep an eye on team stress indicators and adjust workloads accordingly.
- **Set Realistic Goals:** Establish achievable goals and deadlines during peak periods.
- **Promote Efficient Work Practices:** Encourage efficient work methods to maximize productivity without overburdening the team.

Deadline-Driven Development
When management believes that the only path to improved developer productivity is imposing arbitrary, unrealistic deadlines.

26 🍀 Data-Driven Experimentation

Engineering leaders who adopt data-driven experimentation utilize concrete data, team feedback, and creative trials to enhance their teams' performance. They critically assess the overall impact of each test on the array of engineering metrics and alert their teams to any accidental yet significant effects. Success is quantified by results within set timeframes.

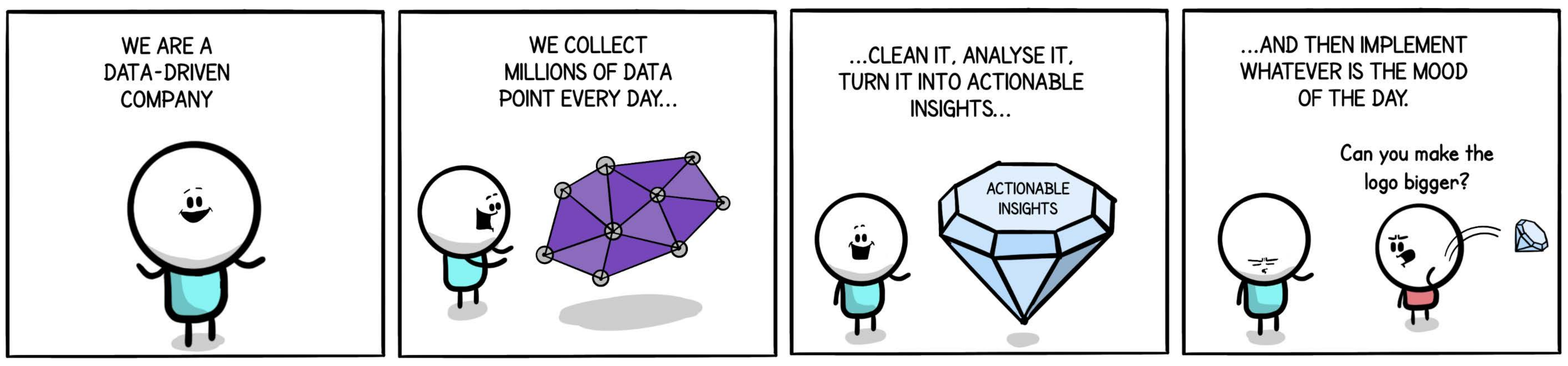
Where to find inspiration for experimentation:

- **Industry Best Practices:** Look to industry leaders and standard practices for potential experimental ideas.
- **Feedback From Teams:** Regularly solicit and use team feedback to identify areas for improvement.
- **Customer Insights:** Utilize customer feedback and usage data to guide experimental focus.
- **Competitor Analysis:** Study competitors' approaches to identify potential areas for innovation.
- **Tech Conferences and Workshops:** Attend industry events for insights into emerging trends and technologies.
- **Academic Research:** Explore the latest academic research for cutting-edge techniques and theories.
- **Online Tech Communities:** Engage with online tech communities for diverse perspectives and ideas.ten

How to encourage it:

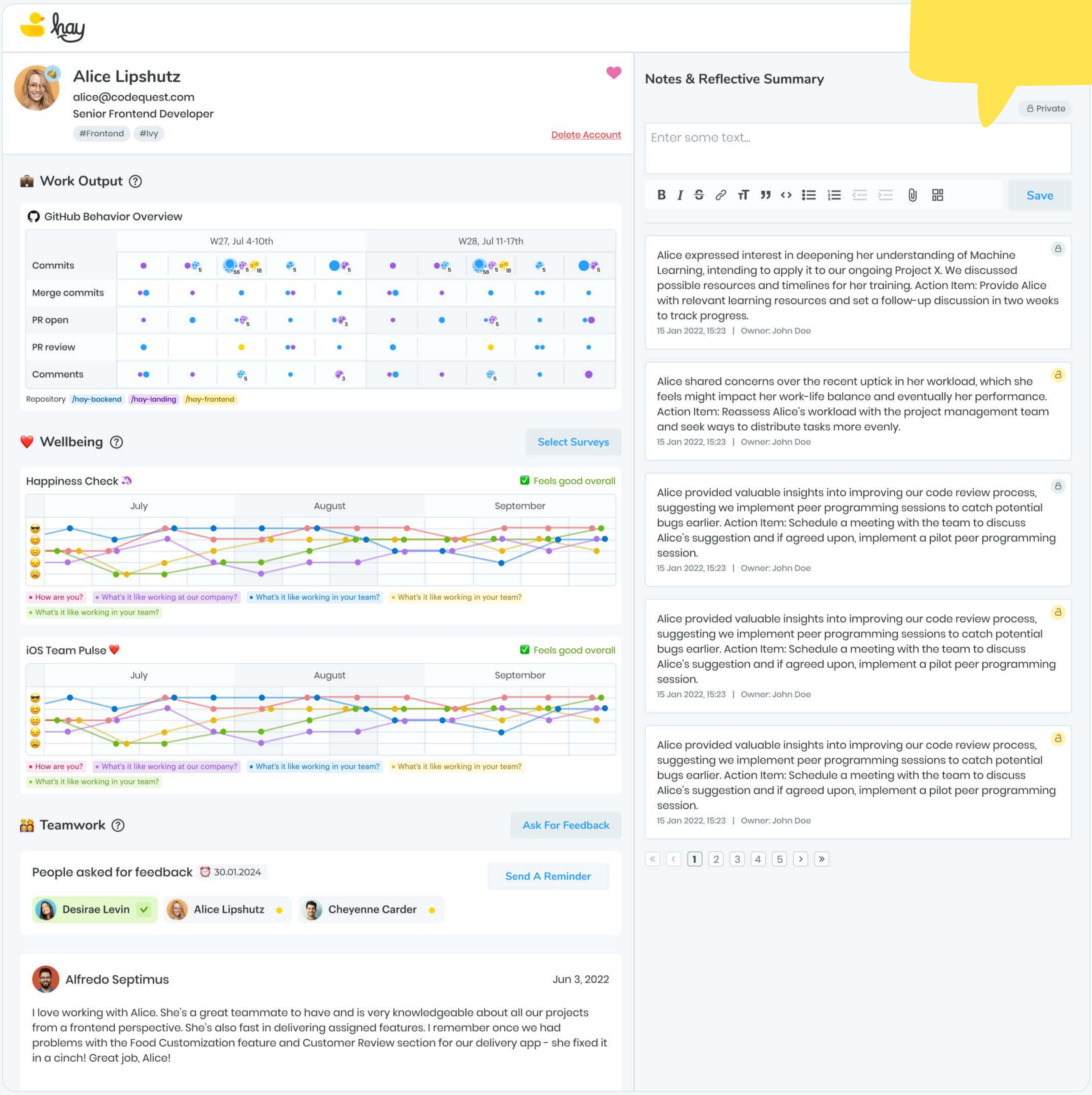
- **Create a Safe Space for Trials:** Foster an environment where experimentation is encouraged and failure is seen as a learning opportunity.
- **Set Clear Objectives and Metrics:** Define specific goals and metrics for each experiment to measure its impact.
- **Provide Necessary Resources:** Allocate the required resources, tools, and time for conducting experiments.
- **Incorporate Regular Reviews:** Schedule regular check-ins to assess progress and pivot as necessary.
- **Encourage Cross-Functional Collaboration:** Promote collaboration across different teams to bring varied perspectives to experiments.
- **Highlight Success Stories:** Share successful experiments widely to inspire and motivate the team.
- **Incentivize Innovation:** Recognize and reward team members who contribute innovative ideas and participate in experiments.

"The goal is to turn data into information, and information into insight."
— Carly Fiorina


WE ARE A DATA-DRIVEN COMPANY
WE COLLECT MILLIONS OF DATA POINT EVERY DAY...
...CLEAN IT, ANALYSE IT, TURN IT INTO ACTIONABLE INSIGHTS...
ACTIONABLE INSIGHTS
...AND THEN IMPLEMENT WHATEVER IS THE MOOD OF THE DAY.
Can you make the logo bigger?
workchronicles.com

Looking for a tool that will help you check in on your developers' wellbeing?

Try HAY Now!



Alice Lipshutz
alice@codequest.com
Senior Frontend Developer
#Frontend #Ivy
Delete Account

Work Output
GitHub Behavior Overview
W27, Jul 4-10th | W28, Jul 11-17th
Commits
Merge commits
PR open
PR review
Comments
Repository /hay-backend /hay-landing /hay-frontend

Wellbeing
Select Surveys
Happiness Check — Feels good overall
July August September
• How are you? • What's it like working at our company? • What's it like working in your team? • What's it like working in your team? • What's it like working in your team?

iOS Team Pulse — Feels good overall
July August September
• How are you? • What's it like working at our company? • What's it like working in your team? • What's it like working in your team? • What's it like working in your team?

Teamwork
Ask For Feedback
People asked for feedback 30.01.2024
Send A Reminder
Desirae Levin ✓ | Alice Lipshutz ● | Cheyenne Carder ●

Alfredo Septimus — Jun 3, 2022
I love working with Alice. She's a great teammate to have and is very knowledgeable about all our projects from a frontend perspective. She's also fast in delivering assigned features. I remember once we had problems with the Food Customization feature and Customer Review section for our delivery app - she fixed it in a cinch! Great job, Alice!

Notes & Reflective Summary
Private
Enter some text...
Save

Alice expressed interest in deepening her understanding of Machine Learning, intending to apply it to our ongoing Project X. We discussed possible resources and timelines for her training. Action Item: Provide Alice with relevant learning resources and set a follow-up discussion in two weeks to track progress.
15 Jan 2022, 15:23 | Owner: John Doe

Alice shared concerns over the recent uptick in her workload, which she feels might impact her work-life balance and eventually her performance. Action Item: Reassess Alice's workload with the project management team and seek ways to distribute tasks more evenly.
15 Jan 2022, 15:23 | Owner: John Doe

Alice provided valuable insights into improving our code review process, suggesting we implement peer programming sessions to catch potential bugs earlier. Action Item: Schedule a meeting with the team to discuss Alice's suggestion and if agreed upon, implement a pilot peer programming session.
15 Jan 2022, 15:23 | Owner: John Doe

Alice provided valuable insights into improving our code review process, suggesting we implement peer programming sessions to catch potential bugs earlier. Action Item: Schedule a meeting with the team to discuss Alice's suggestion and if agreed upon, implement a pilot peer programming session.
15 Jan 2022, 15:23 | Owner: John Doe

Alice provided valuable insights into improving our code review process, suggesting we implement peer programming sessions to catch potential bugs earlier. Action Item: Schedule a meeting with the team to discuss Alice's suggestion and if agreed upon, implement a pilot peer programming session.
15 Jan 2022, 15:23 | Owner: John Doe

1 2 3 4 5

Thanks a Bunch for Coding Along with Us! 💻

Thank you for taking the time to read our exploration of behavioral practices in software development. We hope you found it informative and valuable for you and your dev teams.

In case you'd like to add some points or give us feedback, feel free to reach out to me. And if you'd like to learn more about our engineering course or our Slack integration that helps engineering managers gauge their developers' happiness and satisfaction, check out our site here.

We look forward to accompanying you on your journey and hope to be a part of your future learning experiences.

Warm regards,

Tomasz Korzeniowski

Tomek

tomek@howareyou.work

CEO @ HAY

Handy links:

Blog | Course | Contact Us | HAY Slack Community

Linkedin | Youtube | Twitter | Facebook