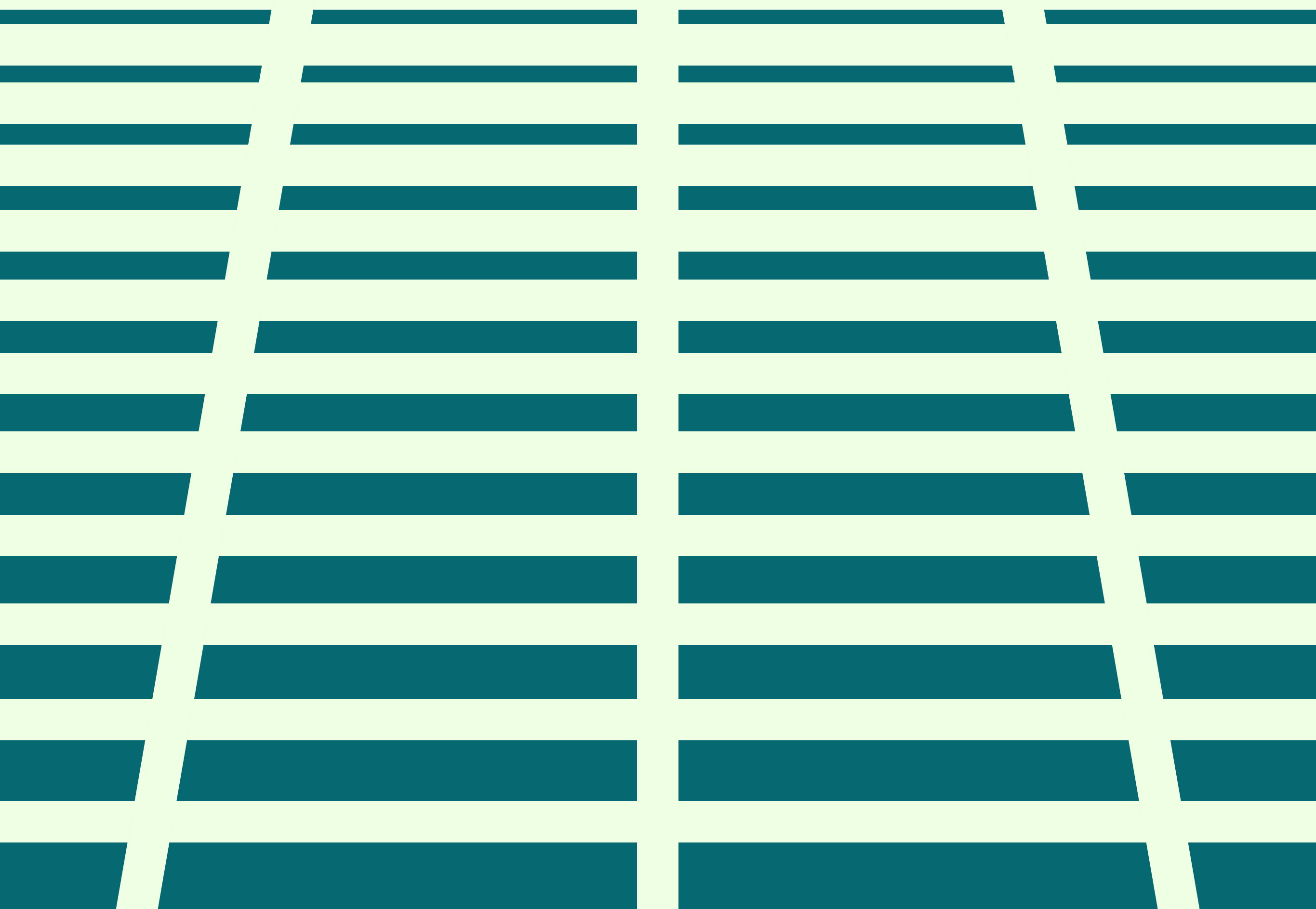
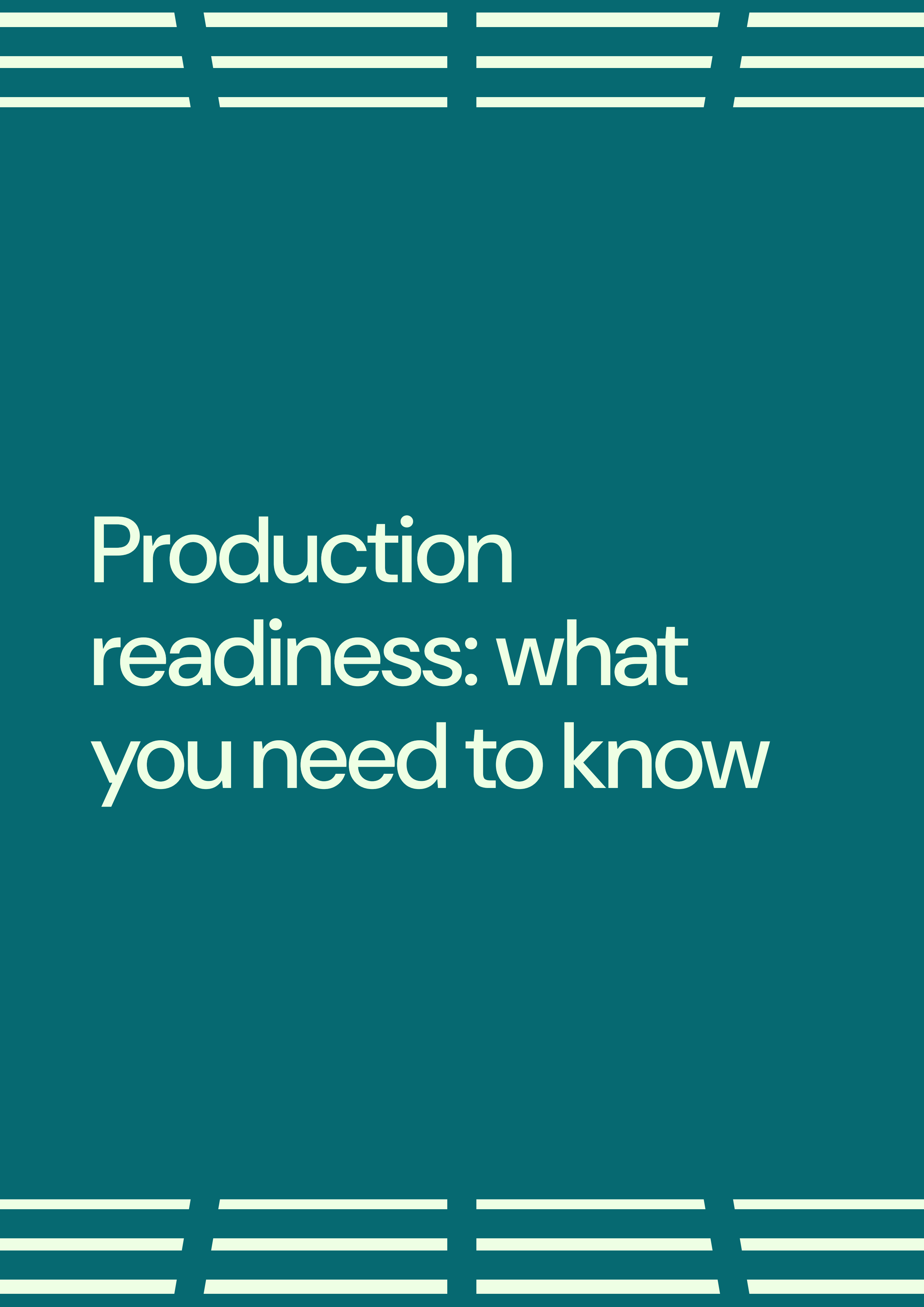


Managing production readiness in a developer portal

 A how-to guide





Production readiness: what you need to know

What is production readiness?

Production readiness means different things to different engineering organizations. However, it can be defined as a process that determines the reliability requirements and levels of a specific software component (such as an API, a microservice etc) operating in production.

The end goal is ensuring the reliability of a service operating in production. The idea was inspired by the [production readiness review process](#) in Google's SRE book.

Another way of thinking about production readiness is '[definition of done](#)', a notion borrowed from product management when all conditions, or acceptance criteria, that a software product must satisfy are met and ready to be accepted by a user, customer, team, or consuming system. In this case, for production readiness, the question is both whether the software entity or service meets the 'definition of done' in terms of its standards compliance, not only when the service is first deployed but also as it moves through the different stages (e.g. promoted to testing, to production and so on) and as its underlying conditions change.

Suppose an engineering team isn't able to ensure production readiness. In that case, it can manifest in many ways, for example: as higher downtime because of a lack of testing or an increased number of incidents due to a lack of integration between incident management tools, or not establishing an upper limit for unresolved issues.

In general, production readiness is difficult to master because developers have two options. The first is to spend a lot of time waiting for SRE reviews when they are setting up the system, which is frustrating and sometimes wasteful. However, in many cases this was the only way to maintain consistency. If developers chose the second path, i.e. pressing ahead and not complying (either on set up or further down the road) production readiness or other standards won't be met. Internal developer portals can solve this issue as they provide a way to balance checks with speed, using automation and self-service. We'll get to that later.

Production readiness:

A process that determines the reliability requirements and levels of a specific software component (such as an API, a microservice etc) operating in production.

Types of production readiness metrics & checklists

[Production readiness](#) is synonymous with software standardization or software onboarding; all are a list of items required to enable software to work well in production.

Each organization has and needs different production readiness metrics and checklists, reflecting different:

- ↘ Business needs (for instance a highly regulated industry with PII); and
- ↘ Technical environments (are the services exposed externally and therefore more vulnerable or are there certain K8s standards to be met, etc.).

A production readiness checklist for a service can encompass numerous factors that ensure its reliability in production. This involves making the service secure, scalable, reliable, and observable, implementing continuous integration and continuous deployment (CI/CD), setting appropriate service level objectives (SLOs), and having disaster recovery and rollback plans. These elements are crucial both when the service is initially launched and as it evolves over time. Let's take a close look at the potential list:

- ↘ Security: ensuring regular vulnerability scans and implementing role-based access controls.
- ↘ Scalability: ensuring the architecture is scalable and can handle heavy loads.
- ↘ Reliability: ensuring the service is highly available.
- ↘ Observability: ensuring the service is monitored and that metrics, logging and tracing are enabled.
- ↘ CI/CD: ensuring the release process is automated and scalable
- ↘ Rollback: ensure the deployment process includes automated rollback capabilities to revert to a previous stable version if necessary.
- ↘ Service Level Objectives (SLOs): ensure SLOs are defined and that compliance with these objectives is monitored.
- ↘ Incident management: ensure that each service has an on-call assigned and an owning team

PR CHECKLIST

Security	Scaleability & reliability	Service Level objectives	Ownership	Documentation
<input type="checkbox"/> Vulnerability scans	<input type="checkbox"/> Scaleable architecture	<input type="checkbox"/> SLOs are defined	<input type="checkbox"/> Service owners	<input type="checkbox"/> README
<input type="checkbox"/> Least privilege policies	<input type="checkbox"/> Monitoring tools	<input type="checkbox"/> SLOs are monitored	<input type="checkbox"/> On-call	<input type="checkbox"/> Documentation
<input type="checkbox"/> Branch protection	<input type="checkbox"/> Freshness			<input type="checkbox"/> Resource definition

By addressing these aspects, a microservice can be considered production-ready, ensuring it meets the demands of its users and maintains reliability throughout its lifecycle.

Not all these metrics need to be tracked for every service, as we'll discuss below. Additional metrics that aren't necessarily part of SRE work can be added, such as FinOps metrics, Kubernetes standards or AppSec standards.

A production readiness checklist is difficult to establish because the various software entities (APIs, microservices etc.) all require different standards that depend on a variety of factors, from their infrastructure and underlying technology to their centrality in the given engineering environment.

Production readiness is continuous

If a service was production-ready when it was scaffolded, will it remain that way over time? The answer is no.

While it's tempting to believe that if you set guardrails and golden paths for developers, production readiness will likely be covered well, this isn't true. Requiring observability or security may be a good first step, but production readiness may change (when requirements change) or degrade over time.



That's why it's important to have the capability to continuously (and automatically) check that services are up to production readiness standards and to automatically prompt engineers to fix services when needed.

Ongoing manual vs automated production readiness checks

Manual checks of production readiness involve manually updating data about software to verify that all necessary criteria are met before a service is deployed or updated or after its deployment. This process typically uses tools such as spreadsheets, manually updated project management software, or Configuration Management Databases (CMDBs). These tools require team members to individually add data that's relevant to production readiness to allow verification that all aspects of production readiness—such as security measures, scalability, reliability, observability, CI/CD processes, SLOs, disaster recovery, and rollback plans—are in place.

While manual checks can be thorough, they are often time-consuming, prone to human error, and may lack real-time accuracy, eroding trust in the system as a whole. Additionally, the manual approach can lead to inconsistencies and delays, especially in complex environments where numerous interdependencies exist.

In contrast, automated checks using scorecards of production readiness leverage the power of internal developer portals which integrate with various tools and systems to continuously monitor and validate the readiness criteria. These portals automate the collection and analysis of data related to the service's health, compliance, performance, and other key metrics. Automation ensures that checks are consistently performed without human error, providing real-time insights and alerts when issues arise. Internal developer portals can automatically enforce policies, trigger tests, and validate configurations, significantly speeding up the process and increasing reliability. This approach not only reduces the manual workload on teams but also enhances the accuracy and efficiency of the readiness checks, ensuring that services are always in a state of production readiness.

 Manual (Add data to spreadsheet/project management software/CMDBs, etc.)	 Automated (Scorecards via internal developer portals)
Data input and collection is time-consuming	Collection and analysis is automated, saving time
Prone to human error	Checks consistently performed without human error
Lacking real-time accuracy, eroding trust	Provides real-time insights and alerts
Can lead to inconsistencies and delays	Automatically enforce policies, trigger tests and validate configurations, speeding up process

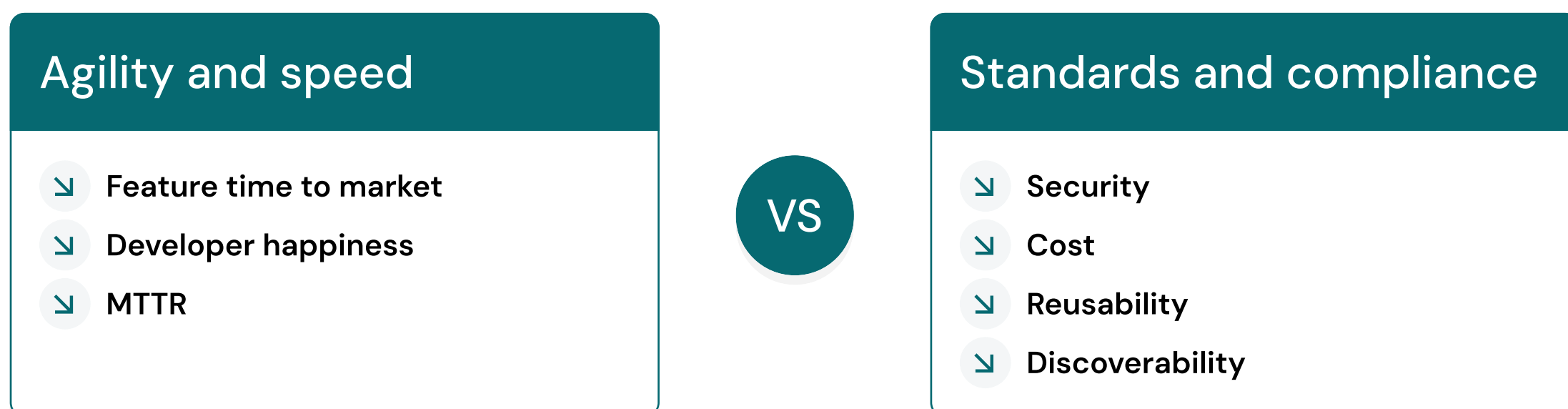
Do all services need the same level of production readiness: the engineering standards dilemma

We all want to achieve high quality software that complies with production readiness and other standards, yet experienced developers know that ‘perfect’ is not achievable, simply because we all have our own time constraints and business priorities. So how can we find the right balance when setting up engineering standards?

Standards are essential. They help prevent messy code, ensure your software can handle growth, and keep technical debt under control. Without proper standards, projects can become difficult to manage.

On the other hand, software moves fast. Rigidity can hold innovation and slow you down– something you want to avoid in a competitive landscape. It's a balancing act: too few standards, and you risk a messy codebase; too many, and you might fall behind.

So, how do you find that middle ground? How do you create standards that promote quality and reliability without sacrificing agility and speed? That's what we'll be exploring as we discuss building a framework for production readiness that works for your team with the help of a developer portal.



The image features a dark teal background with a decorative pattern of horizontal white lines at the top and bottom. The lines are arranged in four columns, with each column containing three lines. The central text is white and reads:

Using a developer portal for production readiness

Tier 1

Critical assets: authentication systems, core business logic, data storage, payment processing.

Tier 2

Major assets: important but non mission critical such as: user interfaces, major API endpoints, and reporting modules.

Tier 2

Minor assets: minor UI elements, experimental features, logging components.

Step 1: Create service tiers and set production readiness standards for them

Not all parts of your software are created equal. Some assets are mission-critical, impacting core functionality, security, or performance. Others are less essential, focusing on peripheral features or experimental components. Recognizing this difference is key to creating a flexible and effective standards framework.

A common approach is to tier your software assets based on two factors:

- ↘ **Criticality:** How vital is this asset to the overall functionality and success of your software? Does its failure have a significant impact on users or your business?
- ↘ **Risk:** What's the likelihood of this asset failing or causing issues?

By evaluating these factors, you can categorize your assets into different tiers, each associated with specific standards:

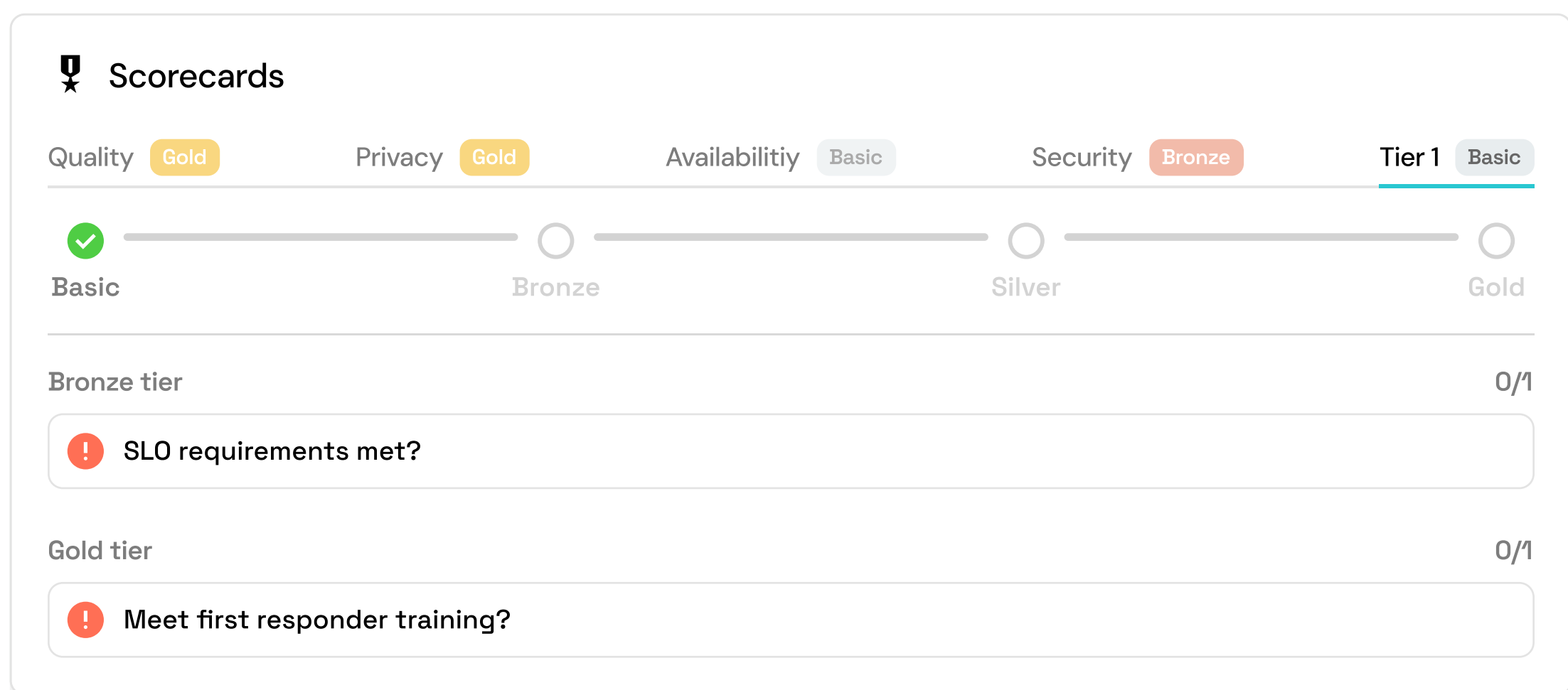
- ↘ Tier 1: Critical assets
 - These assets are the backbone of your software. They directly impact core functionality, security, or performance. Examples: Authentication systems, core business logic, data storage, payment processing.
- ↘ Tier 2: Major assets
 - These assets are important but not mission-critical. They contribute to significant features or functions. Examples: User interfaces, major API endpoints, and reporting modules.
- ↘ Tier 3: Minor assets
 - These assets are less critical to core functionality and have a lower risk of failure. They often involve small, non-essential components. Examples: Minor UI elements, experimental features, logging components.

Step 2: Set the right scorecards for the different tiers

Once you've categorized your software assets into tiers, the next step is to define the appropriate standards for each tier. A best practice approach is to establish a baseline set of requirements for your lowest tier assets. This ensures a minimum level of quality and consistency even for less critical components.

In an internal developer portal, standards are reflected by scorecards. A scorecard is a way to measure and track the health and progress of each service and application within your software catalog. Scorecards establish metrics to grade production readiness, code quality, migration quality, operational performance, and more.

As services move up tiers, you can add more scorecards. Tier 2 assets will have a bigger set of standards, while Tier 1 assets will demand the most rigorous checks and balances, all of which will be reflected in the standards. For example, you can check that all Tier 1 services have met both SLO and first responder training requirements.



In the portal, you can establish scorecards for any domain, with tiers weighted on criticality and risk.

You might be tempted to add many software components to Tier 1, but our advice? Don't. Managing Tier 1 assets is costly—they need more testing, monitoring, and longer review cycles. Save this tier for the truly critical stuff.

Lower-tier assets are much easier to tweak and improve, giving you the flexibility to innovate. Make sure anything in Tier 1 goes through a solid approval process to confirm it's really that essential.

By using this tiered approach, you can keep your high standards where they matter most, while staying agile

with the less critical parts. When assets change a tier – for instance, graduate into a mission critical part of your application, you can use the portal to change the scorecards that apply to them.

Change tier

✔ Approve
✘ Decline

Details

Status	Waiting for approval	Port user info	VB GitHub admin
Service	🔄 Cart	Start	a few seconds ago
Form input	See details	End	▬
Payload	See details	Duration	▬
Response	▬	Job links	▬
Summary	▬		

When you want to change a service to a higher tier, you can use the portal to request manual approval, and then automatically apply the new standards

Step 3: Monitor your top tier assets

We’ve talked about how critical it is to keep an eye on Tier 1 assets because any downtime can spell disaster. Maintaining high standards for these services should be your top priority. Think of your standards as the last line of defense against failures, so close monitoring is a must to prevent any degradation.

- ▾ Use dashboards; To stay on top of things, you can set up a dedicated dashboard for your top tier assets. This could act as an SRE dashboard (or a DevSecOps dashboard, depending on the standards you’re tracking). This will help you increase your confidence level by tracking compliance with company standards in real-time.
- ▾ Automated alerts: Implement Slack/Teams notifications to get instant alerts if any scorecard for a Tier 1 asset starts to degrade. These notifications should go to the central monitoring team, the engineering leadership and the service owners, to ensure that everyone responsible is in the loop and can act quickly.

System-level summary

Team	Title	Quality	Privacy	Availability	Tier 1
TS	▬	▬	▬	▬	▬
TN	▬	▬	▬	▬	▬
TB	▬	▬	▬	▬	▬

This dashboard in Port shows you the distribution of scorecards across software assets

Step 4: Drive actions from standards – alerts and automations

Having standards is great, but they're not just there to look good on a fancy dashboard—they need to drive action.

Every rule that isn't met should automatically generate a task for the asset owner. Whether it's missing documentation, a misconfiguration detected, or any other compliance issue, it should trigger a clear, actionable task.

Don't forget to assign deadlines to these tasks. It will ensure that issues are resolved quickly and are no longer a problem. Monitor the completion of these tasks to make sure they are taken care of in a timely manner.

You can do this using automations in your developer portal.

Tier 1 tasks to reach the gold level for ser...

Open issue

Matarp13

Tier 1 tasks for the service: Wish list
This issue contains all sub-tasks needed to be completed for Wish list to reach the gold level in the tier 1 socercard.

Sub tasks

Tier 1 open issues

Port team

- PL Platform ▼
- AR API request ▼
- CO Codespaces ▲

Title	Service	Status
Tier 1 tasks to re...	New-service-926	Open
Tier 1 tasks to re...	MyNewService908	Open
_____	_____	Open

Step 5: Use initiatives when standards evolve

Your standards are not static; they are evolving. There will come a time when you need to add a new requirement. When this happens, update the relevant scorecard and create an initiative page so everyone can track progress (initiatives will also remind developers using various automation features in the portal).

Initiatives align developer workflows to business KPIs. They contain a collection of related scorecards that drive developers to adopt practices and tooling that support those goals. For example, an 'improve reliability' initiative could contain scorecards for crash-free releases, mean time to recover (MTTR) and end-to-end testing coverage. A portal's dashboards can be used to communicate initiatives and to easily track them by developer, team and service.

Typically, you'll include some key information:

- An overview of the initiative and its goals.
- Progress updates from each team towards meeting the new requirement.
- A detailed list of assets that aren't compliant, allowing you to nudge the owners.

Flexibility is important, so consider allowing teams to request extra time if needed. Sometimes a given service may not be compliant and the developer just needs some additional time to become compliant. However, this needs to be managed carefully as you risk further issues down the line. For example, if a package has been deprecated teams should request exceptions (using self-service forms in the portal) if they want to ask for some more time. Manual methods of identifying such packages and alerting owners are more time-consuming and may lead to missed deadlines. Using a portal, you can use scorecards to check that services are up-to-date, ensure components have been migrated, and make sure that services comply with new standards. You can also enforce a manager approval process for exemption requests and keep track of them to know if they've been approved, why, and what the new deadlines are.

⚠ Services pending migration

Title	Tier	Version	Migration
📦 Pricing	Tier 1	19.8	21m ago
📦 Shipping	Tier 1	19.8	In a month
📦 Analytics	Tier 2	19.8	In a month
📦 Notifications	Tier 3	19.8	In a month

★ Services NodeJS versions

• 22 • 19.8 • 21.1

⚠ # Services not migrated

4

👥 Progress by team

Team	Title	Node JS version updated?	Property
👤	—	75%	—
👤	—	50%	—
👤	—	0%	—
👤	—	50%	—

🕒 Extra time pending approval

Action	Entity	Status	Post user info	Starts	End
—	—	Waiting for approval	...	—	—


1 result


Step 6: Use self-service actions in the portal to quickly react


With Tier 1 assets in particular, you want to be able to react quickly when there is an issue. Each use case and organization has its own processes; some organizations, for example, choose to lock deployments for services that fall below certain standards. Alternatively, you may want to open an incident or simply nudge the owner to address a problem. Self-service actions enable developers to perform tasks themselves without filing tickets or waiting for other teams. These actions can be created easily in the portal by platform engineers.


Being able to perform all these actions from one central place in the portal, in context while having all the relevant information at your fingertips, is really powerful. This centralized approach ensures that you can respond fast and effectively to any issues, minimize downtime and maintain high standards.

⚡ What should we do?

-  Lock service ⚡

-  Nudge reviewers ⚡

-  Create incident ⚡

-  Change tier ⚡

👤 Tier 1 requirements

Port team	Title	SLO requirements	Meet first
TS	---	33%	0%
TN	---	33%	0%
TL	---	33%	0%
TP	---	33%	0%
TB	---	33%	0%

Quick actions directly in the portal help you react fast to service degradation

Conclusion

A developer portal is great for closing the loop on managing engineering standards. It helps you set up tiering, associate assets with the right standards, and track compliance. More importantly, it drives action by automatically generating tasks for non-compliance and allows for quick reactions when issues arise. By consolidating all these functions in one place, a portal ensures your team can maintain high standards while staying agile and responsive, ultimately leading to better software and a more efficient development process.

Want to see how production readiness works on Port?

Try our live demo