OverOps

The Complete Guide to

Delivering Reliable Software in the Enterprise

Tali Soroker

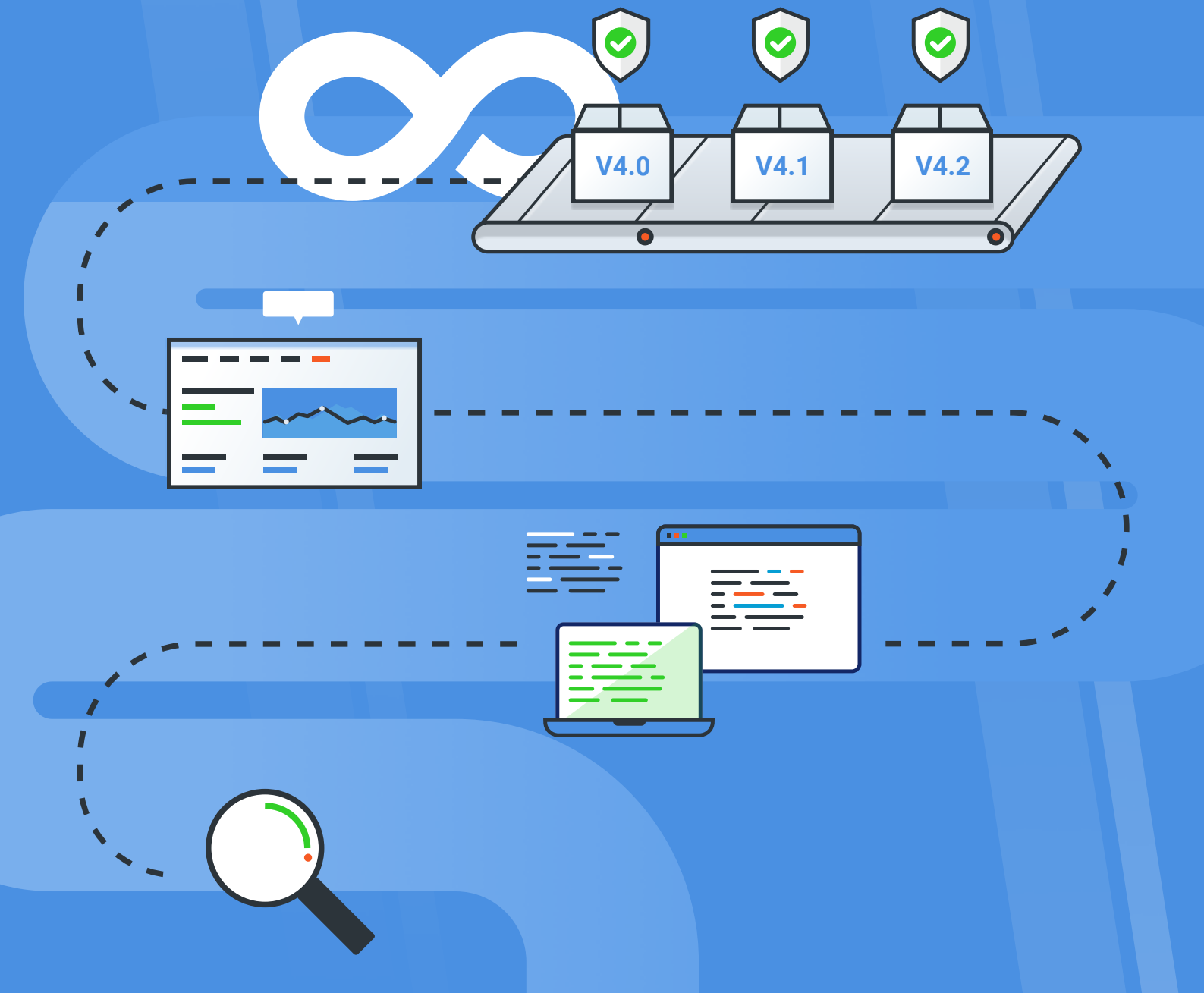


Table of Contents

Introduction

Enterprise organizations are beginning to realize that new strategies are needed to ensure consistent delivery of quality code in software products. We've seen the emergence of CI/CD practices and the rise of automation help drive innovation through faster code delivery. But the target is no longer just shipping code faster – it's also closing new and widening reliability gaps.

To help organizations better manage this balancing act between agility and reliability, OverOps developed the below reliability maturity model. Based on conversations with hundreds of engineering organizations in various stages of their reliability journey, OverOps identified four key phases of reliability maturity marked by common characteristics and challenges.

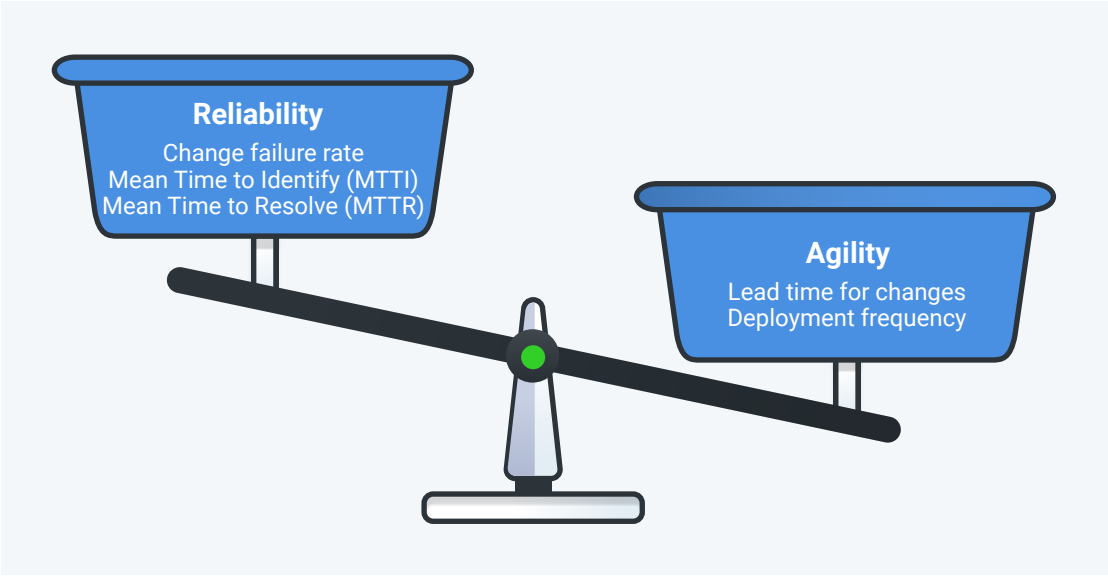
This model was built to provide a North Star for where to invest resources and focus your energy as you strive not only to achieve but to maintain reliability.

Using this framework, organizations can:

- Identify strengths and gaps in their reliability strategy
- Prioritize and address their most critical reliability challenges
- Understand which processes and tools they need to invest in
- Remove engineering roadblocks that hinder productivity and hurt customer experience
- Secure executive buy-in for reliability initiatives

Chapter 1: The Agility-Reliability Paradox

Development and IT Ops teams commonly find themselves in a game of tug-of-war between two key objectives: agility (i.e. driving innovation) and reliability (i.e. maintaining stable software).



To drive innovation, we've seen the emergence of CI/CD practices and the rise of automation. The investment in agility has provided the means to release code faster – the average release frequency at the enterprise level is about once a month (with many releasing more frequently) – but it's also become increasingly more difficult to ensure software reliability.

"Our executives were so focused on delivering new features that it prevented us from reaching our five nines availability target."

- Production Support Lead, Fortune 500

As a result, many IT organizations experience significant code-level failures on a regular basis. These issues go beyond immediate loss of service. They hurt productivity, derail product roadmaps and jeopardize customer experience – all while increasing infrastructure expenses.

Recent estimates suggest that the cumulative cost of poor quality software in US organizations – including costs from technical and legacy debt, cancelled projects, software failures, and time spent fixing defects – reached $2.8 trillion during the last year alone and will grow substantially by 2020.

"We can't deploy without introducing at least one major issue."

- SRE Director, Fortune 500

Derek D'Alessandro, a catalyst for DevOps adoption and technological change, has some unique insights regarding the effects of changes on our systems. **In a post** examining the different modes of change that teams can adopt, he says:

"It is easy to see the benefit of individual changes. The trick to incremental change is measuring and planning out lots of small changes to ensure they all work together and are all headed in appropriate directions. This is an area in which we often create divergence in our systems, technologies or processes. The divergence sneaks up on us and we only discover it when things become unstable and start to topple, or cross integrations are too complex because it's not built with a solid architecture."

The point that Derek makes about adopting change aligns well with our discussion of the Agility-Reliability Paradox. Namely, shipping new features and driving innovation is great, but it can be detrimental to the reliability of our systems without continuous efforts to maintain it.

Basically, we need to find a way to move fast WITHOUT breaking things...

Chapter 2: The Pillars of Continuous Reliability

To properly introduce this framework, we need to first define Continuous Reliability and the components needed to achieve it.

Continuous Reliability

NOUN

The notion of balancing speed, complexity and quality by taking a continuous, proactive approach to reliability across the software delivery lifecycle. It is achieved through data-driven quality gates and feedback loops that enable repeatable processes and reduce business risk.

Achieving Continuous Reliability means not only introducing more automation and data into your workflow, but also building a culture of accountability within your organization. This includes making reliability a priority beyond the confines of operations roles, and enforcing deeper collaboration and sharing of data across different teams in the software delivery lifecycle.

By creating a feedback loop between phases of software development and delivery, organizations can accelerate error resolution and establish data-driven quality gates that prevent issues from reaching production.

The ability to develop quality gates and feedback loops is dependent on to two key pillars of Continuous Reliability:

1. Meaningful Data

Reliability initiatives most often succeed or fail based on the quality of the data that engineering teams rely on. This quality is based both on the methods used to capture the data, as well as the depth of context it provides.

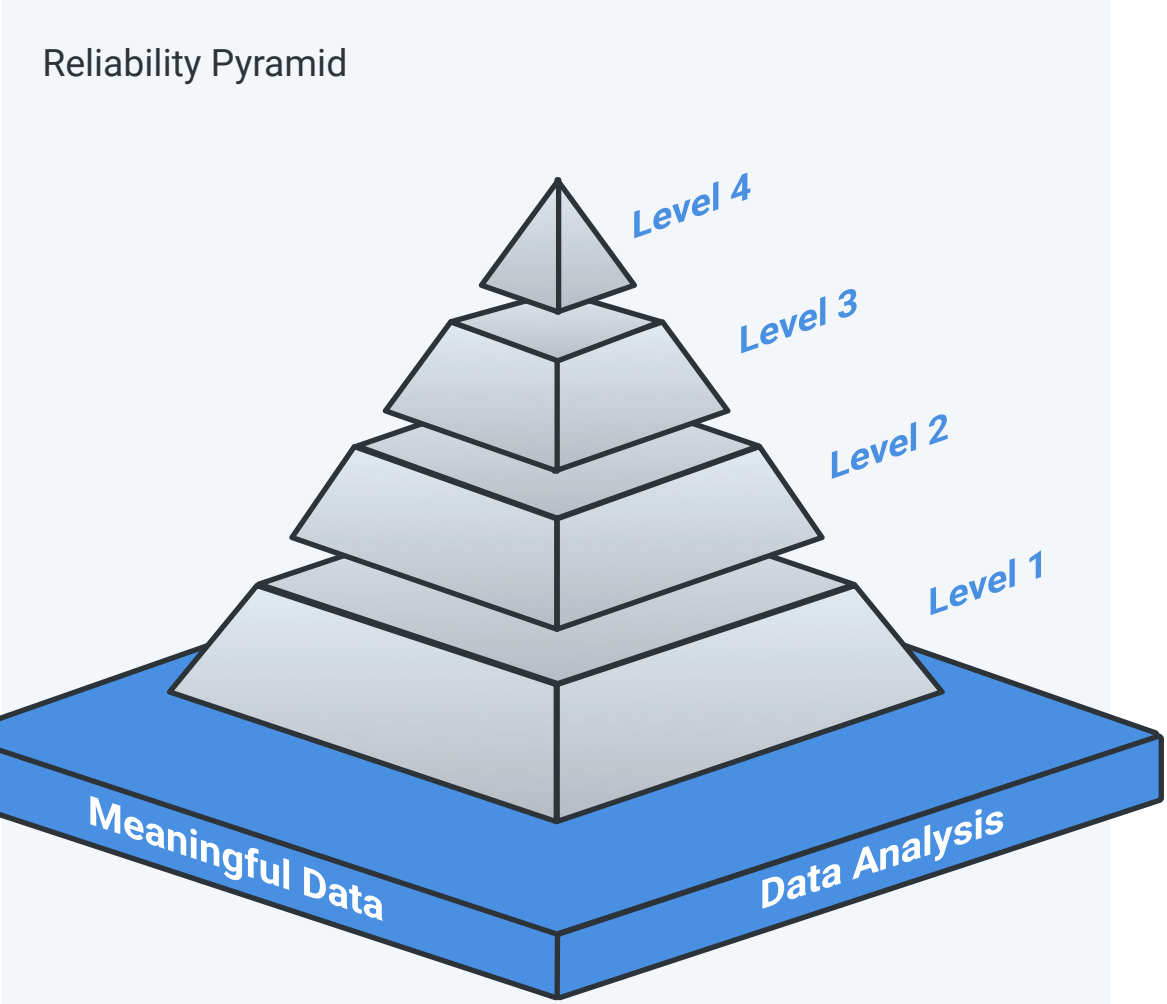
Many organizations struggle with capturing not only every technical failure that occurs, but also enough data about the moment the failure occurred to paint a full picture of what caused the problem. This includes insight into which deployment or microservice an error came from, historical context around when an issue was first or last seen, correlation with system metrics, insight into the source code and state of related variables and more.

Unfortunately, it is common for engineering teams to rely on manual and shallow data sources like log files as their primary source of troubleshooting an error, which often demand an unrealistic amount of foresight and fail to show the full picture.

2. Data Analysis

Even if you can capture all the data in the world, it's only as meaningful as your ability to understand and leverage it in a timely manner. In earlier stages of reliability maturity, analysis is done as an isolated function and reaches a glass ceiling that's set by the ability to collect meaningful data. This requires the ability to capture data in real-time, as well as to apply machine learning and algorithms that surface patterns and help focus your efforts on the issues that matter most.

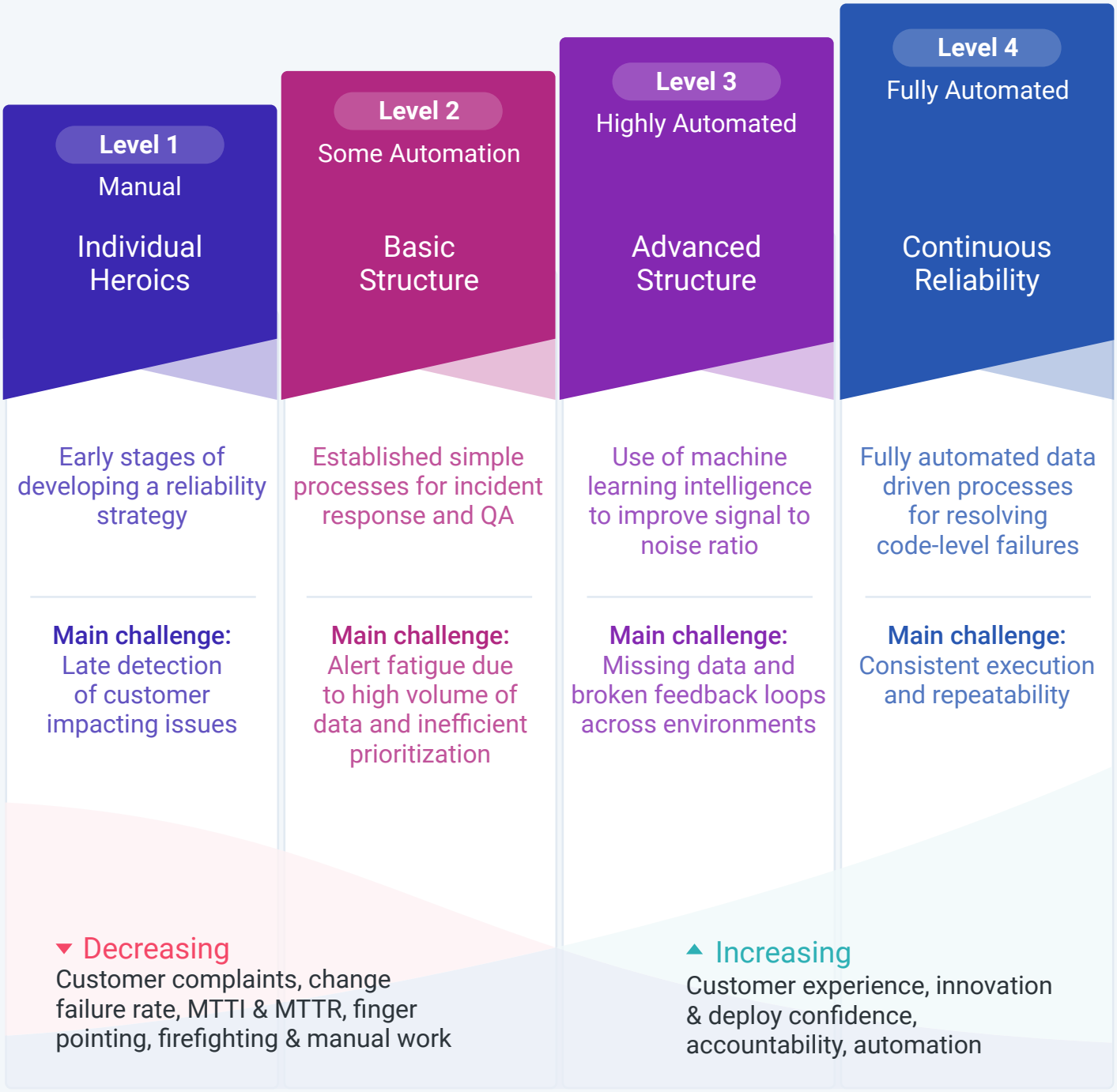
Organizations that successfully achieve Continuous Reliability are able to effectively prioritize issues based on code-level data and inform future data collection through real-time analysis and reinforced learning. This serves as a foundation to define custom quality gates that stop poor quality releases from ever making it to production.



Chapter 3: The Continuous Reliability (CR) Maturity Model

The Continuous Reliability Maturity Model is comprised of four levels that align with common patterns of obstacles and pitfalls organizations encounter related to capabilities in data collection and analysis. Below we break down the characteristics and challenges that define each level and provide recommended next steps to help teams improve.

Continuous Reliability Maturity Model



The Four Levels of Reliability Maturity

Level 1 - Individual Heroics

Organizations at this level are just beginning their reliability journey. This stage is marked by the initial establishment of reliability practices – often leaning toward manual and reactive processes with loose structure. Often, teams at this stage are relying on ad-hoc and inconsistent strategies to solve technical issues. Visibility is a major challenge, and most code quality problems are only addressed if a customer complains.

Individual Heroics Characteristics

Overview:

Ad-hoc and manual processes for solving technical issues; early or experimental stages of prioritizing and formalizing reliability strategy; limited visibility into application errors and their root cause.

Primary Challenge:

Manual and reactive processes and limited visibility into what's happening within your applications and services, resulting in late identification of customer impacting issues.

Next Steps:

- Invest in best practices and a monitoring ecosystem that increase visibility into your system
- Begin establishing best practices for addressing technical incidents
- Clarify roles and responsibilities as they relate to ensuring application quality and reliable operations in production

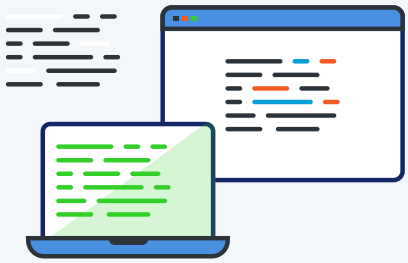
Key Metrics to Advance to Level 2:

- Reduction in MTTI
- Reduction in # of tickets closed as "could not reproduce"
- Increased availability

The Four Levels of Reliability Maturity

Level 2 - Basic Structure

At this stage, teams have established a basic structure with some troubleshooting processes. Application visibility increases as huge amounts of data become accessible through expanded tooling, but the ability to separate the signal from the noise becomes a main challenge as teams seek to better understand which issues have the greatest impact on reliability.



Basic Structure Characteristics

Overview:

Established processes for incident response and QA; some automation across the SDLC; marked reduction in the number of incidents reported by customers; increased visibility into your system through tooling and processes results in higher volumes of alerts.

Primary Challenge:

Inefficient prioritization resulting in alert fatigue and a need to reduce noise.

Next Steps:

- Introduce anomaly detection capabilities through machine learning
- Document and refine your organizations alerting, escalation and issue resolution priorities
- Optimize on-call procedures and implement a culture of code accountability

Key Metrics to Advance to Level 3:

- Reduction in # of false-positive alerts
- Reduction in # of incidents reported by customers
- Further reduction in MTTI

The Four Levels of Reliability Maturity

Level 3 - Advanced Structure

At this point, teams are better able to focus their efforts on issues that matter. They have anomaly detection capabilities that help to manage alert fatigue. But despite the seemingly endless amounts of data being collected, issues are still missing context, and errors still make it to production. Technical debt remains a mystery.

Advanced Structure Characteristics

Overview:

Reduced alert fatigue due to applied intelligence and added context to existing data; automated processes for routing issues to the right people at the right time; increased confidence in processes, tools and team structure; still experience critical production issues that catch you by surprise and you struggle to resolve.

Primary Challenge:

Broken feedback loop between production and pre-production due to data blind spots (unknown unknowns).

Next Steps:

- Invest in new data sources and analysis capabilities to cover the unknown aspects of how your applications behave
- Incorporate learnings from production into the QA process for a more proactive approach to reliability
- Improve cross-team collaboration

Key Metrics to Advance to Level 4:

- Reduction in MTTR
- Customer Experience KPIs (e.g. NPS, customer satisfaction, end-to-end transaction times)
- Reduction in MTTI for previously undetected issues

The Four Levels of Reliability Maturity

Level 4 - Continuous Reliability

This is the most mature stage of reliability, but the work is never done. At this level, teams have access to nearly all of the relevant data they need to troubleshoot issues quickly and to monitor reliability based on collected metrics.
Quality gates are set up between the stages of development to automatically block the progression of unreliable code. Feedback loops are also streamlined to ensure that software quality is not only stable, but improving over time and easy to measure.
Main challenges at this stage are consistent execution by team members based on the available data and analysis capabilities.

Continuous Reliability Characteristics

Overview:

Established and highly-automated processes; ability to capture deep contextual data that fuels feedback loops between teams and stages of software development and delivery.

Primary Challenge:

Maintaining consistent delivery

Next Steps:

- Continue to optimize your reliability processes through detailed post-mortems and data-driven feedback loops between stages of your SDLC
- Apply learnings across the organization

Key Metrics to Measure Success:

- Customer Experience KPIs (e.g. NPS, customer satisfaction, churn)
- Business KPIs (e.g. application revenue, user acquisition, delivery of new products)
- Reduction in change failure rate
- Additional reduction in MTTR+MTTI

Chapter 4: OverOps' Unique Data Capabilities

As with many other industry trends, advancing towards Continuous Reliability takes a strong combination of people, processes AND tooling. There's no single be-all end-all tool that can take an organization from Level 1 to Level 4 just by adding it to their tool stack. Many of the 'Next Steps' from above - implementing best practices, clarifying roles and responsibilities, improving cross-team collaboration - focus on building a culture of accountability across the organization.

That being said, data collection and analysis are the two key pillars of Continuous Reliability and strong capabilities in both are crucial to achieve high-level software reliability. OverOps is a uniquely powerful reliability tool due to our ability to capture, analyze and present code-level data for every error and slowdown.

With that, here's a breakdown of the four key types of data OverOps captures and why they're critical to advancing in the journey towards Continuous Reliability:

1. Code Metrics

Before you can effectively prioritize and fix critical code-level issues, you first need visibility into exactly which issues are occurring. At the most basic level, OverOps automatically captures 100% of events happening within your application in both test and production – even those missed by your logging framework or APM tools.

This includes:
- Logged errors and warnings
- Uncaught and swallowed exceptions
- Slowdowns and APM bottlenecks

With OverOps, you no longer need to rely on logs and developer foresight into which events to capture, what to include in a log statement, or how to analyze it.

On top of detecting every event, OverOps applies a layer of intelligence to automatically prioritize all events based on severity so your team can focus on the issues that matter most. Taking into account things like if an error is new, when it was first and last seen, how many times it occured and if there has been a sudden increase, OverOps is able to mark errors as severe based on criteria such as if a new or increasing error is uncaught, or if its volume and rate exceeds a certain threshold. It considers established baselines and averages to pinpoint anomalies and immediately notify DevOps and SRE teams of events that require immediate resolution.

2. True Root Cause

Many APM vendors will tell you that they provide the root cause of an issue, including "code-level" insights. What they actually mean is that they provide you with a stack trace. Stack traces, while useful, only help identify the layer of code where an issue occurred. From there, you're left to your own devices, including spending time manually digging through shallow log files to find context that can help you reproduce the issue.

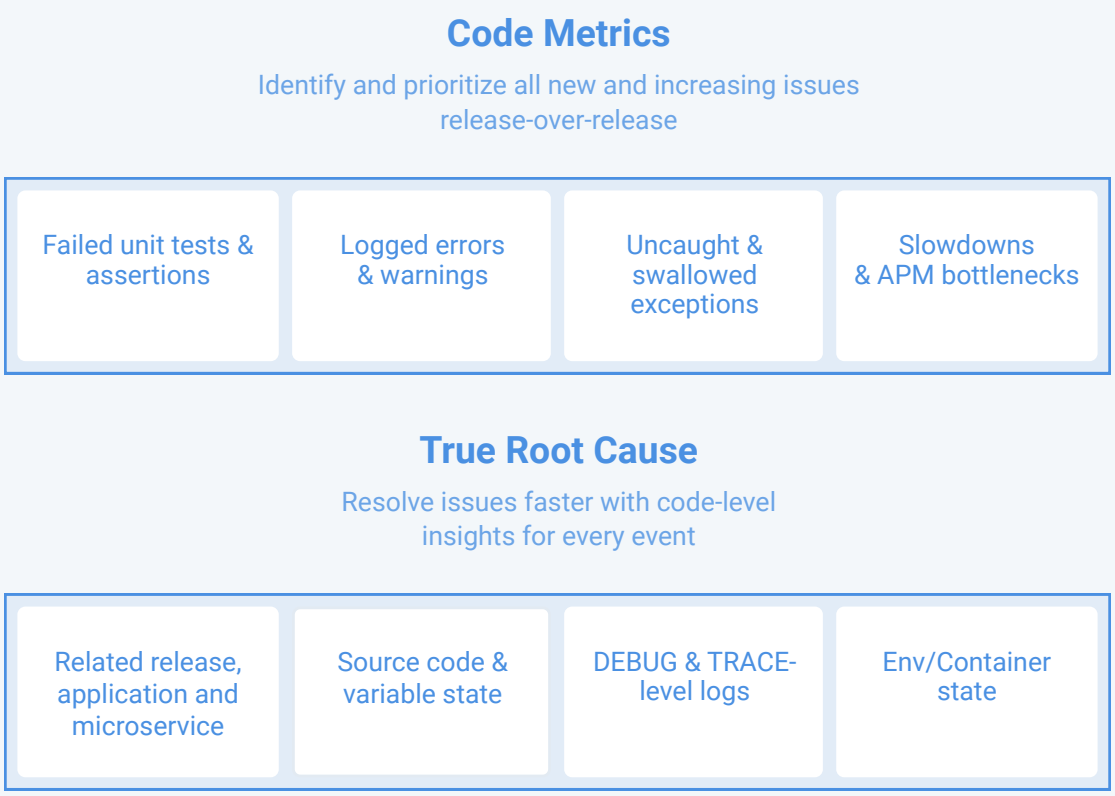
OverOps helps you go beyond the stack trace, capturing deep data, down to the lowest level of detail – without dependency on developer or operational foresight.

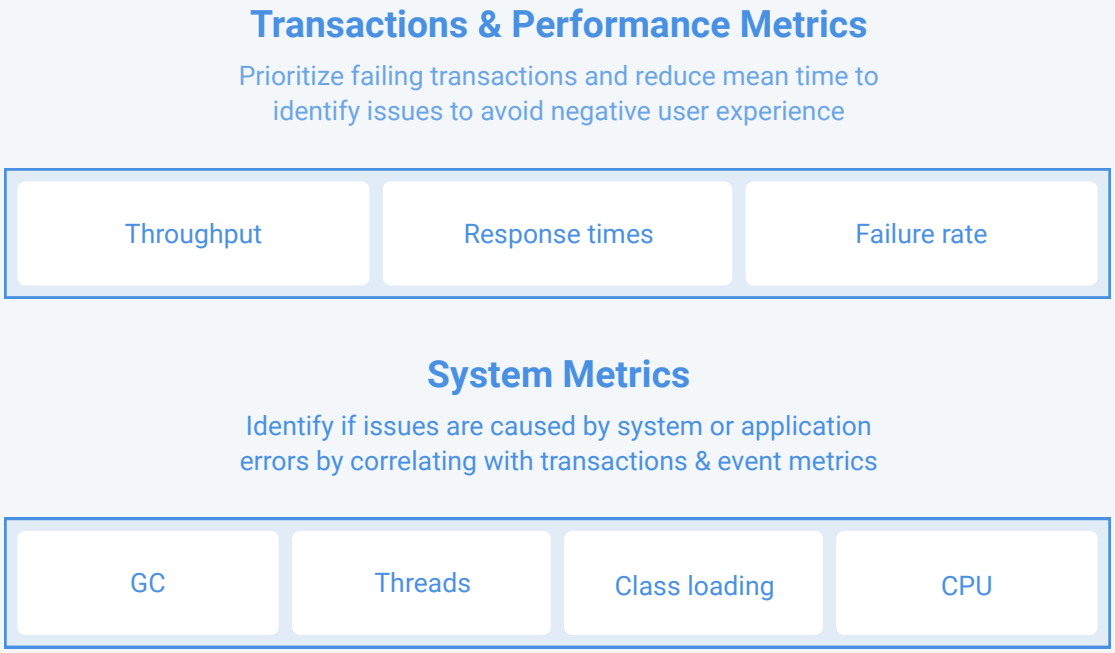
This includes:

- The source code executing at the moment of the incident captured directly from the JVM
- The exact offending line of code
- Key data and variables associated with the incident
- DEBUG and TRACE Log statements
- Env/OS/Container State
- Ability to map Events to Specific Applications, Releases, Services, Etc.

Check out our **recent blog** from OverOps Principal Solutions Architect, Karthik Lalithraj, where we take a deep-dive into the seven key components that make up True Root Cause and why OverOps is the only tool that captures the context needed to effectively troubleshoot issues.

What types of data does OverOps capture?





3. Transactions & Performance Metrics

In the context of software development and reliability, a transaction is a sequence of calls that are treated as a unit, often based on a user-facing function. When a transaction fails, customer experience is often impacted, so it's important to be able to identify and prioritize these failures in the context of the transactions that they impact.

OverOps captures data about every transaction failure, ranging from how many times it happened, to how many transactions failed, to the response time of the transaction. Using insights from the code events we mentioned above, we can determine the success of a transaction by correlating errors, exceptions and slowdowns within a given timeframe and surface this data to our users.

These performance metrics include things like throughput, or the number of transactions that occur during a given period of time, and response time baselines. The ability to capture data about application performance is critical to understanding what your end users are experiencing, as well as correlating related events that may help with identifying the root cause.

4. System Metrics

OverOps focuses on data at the code level of your application, but we recognize the importance of correlating code-level failures with other aspects of your system. For example, what impact did your latest deployment have on CPU/memory utilization? Are there any blocked threads related to this failure? Was this CPU spike caused by the application?

Through the OverOps reliability dashboards, you can correlate events, transactions and performance metrics to things like Garbage Collection, Threads, CPU, Class Loading and Memory Consumption, giving you a more comprehensive view into dependencies indirectly related to your application.

The Next Step in Your Reliability Journey

As organizations progress in their reliability maturity, they automate more processes, increase their signal to noise ratio, and improve team culture. This translates to productivity gains and a better customer experience, ultimately improving the overall bottom line for the business.

The Continuous Reliability Maturity Model is not a silver bullet for solving your reliability woes, but helps teams organize priorities and set actionable objectives. Using this model, engineering leaders can chart a course to reach their goals for reliable and efficient execution. Getting started with this model doesn't have to be overwhelming.

Ask yourself the following simple questions to get a sense for where your team is at on your journey and immediately orient yourself toward the next level of your progression:

Visibility

Does my system have visibility gaps? Am I able to capture 100% of events? What types of issues am I not able to detect?

Accountability

What role does each member of my team play in achieving Continuous Reliability? How well do we collaborate across the SDLC?

Efficiency

How much time do our developers spend debugging? How much unaddressed technical debt do we have?

Prioritization

Do we have a good alerting system – are we receiving too many, too few or the right amount? What criteria do we use to identify which issues need our attention most?

Measurement

What metrics are we using to assess reliability? Does my team have clear metrics that define reliability for our application and, if so, do those metrics take into account the customer experience?

Business Impact

Do we have SLAs? If no, are we ready to define them? If yes, are we meeting them? How frequently are we experiencing customer-impacting issues? How much insight do we have into the financial impact of errors?