Lightrun

Lightrun's Economic Impact on Enterprise Logging & Observability Costs

DECEMBER 2022

Key Findings - Details

Lightrun's Economic Impact

Observability Cost Savings	$466,278
Value of Developer Productivity Improvements	$3,873,480
Value of MTTR Imrpovements	$1,644,000
Lightrun Costs	$360,000
Total 3-Year Cost Savings	**$5,623,723**

Cost Savings

Total Cost Savings (%)	31%
Total Cost Savings ($)	$155,426
Payback Period	< 5 Months
Three-year ROI	**248.1%**

Developer Productivity

Total Improvement in Productivity (%)	21%
Total Cost Savings ($)	$3,873,480
Developer Hours Reclaimed	59,592 Hours
Payback Period	< 2 Months
Three-year ROI	**1434.6%**

MTTR (Mean Time To Resolve)

Average Reduction in MTTR (%)	35%
Average Incremental Increase in Revenue ($)	$1,644,000
Payback Period	< 7 Months
Three-year ROI	**609%**

Contents

Introduction

Lightrun is a Developer Observability Platform that enables developers to securely add logs, metrics and traces to live applications in real time, on demand. No hotfixes, redeployments or restarts required.

The following document is based on Lightrun's own customers' experience, internal benchmarking and industry data, and is derived from a thorough investigation on the logging workflows of modern engineering organizations, as well as a careful analysis of the impact of implementing Lightrun at organizations in various levels of scale.

The impact of Lightrun on enterprise logging costs can be roughly summarized according to three different perspectives:

1. **Cost savings**
2. **Three year ROI**
3. **Payback period**

An easy, "back-of-the-envelope" way to quantify the actual cost in each perspective is by looking at the core challenges that businesses encounter when it comes to logging. Specifically, we'll explore three main challenges, and show how they can be quantified in each of the perspectives listed above:

1. **Licensing & Infrastructure Cost**
2. **Developer productivity**
3. **Mean-time-to-resolve (MTTR)**

CHAPTER 1

The Cost of Logging

There are a few characteristics of logging that create problems for software-driven businesses that need to move fast and innovate.

The first is that—with traditional static logging—one faces a number of key limitations. The way around them is only ever to add more logs. However, when the main solution is to add more logs, log volumes tend to grow unexpectedly and exponentially.

The second characteristic is around the fact that creating and deploying a log is not a frictionless process.

Since one can only add logs in development, adding more logs requires going through a whole development cycle to test each iteration of said new logs. This can create delays and interruptions and creates a tendency, very common among modern developers, to overlog applications in order to ensure that every possible contingency is accounted for.

This delays any attempts to resolve bugs, outages and technical issues because of the need to redeploy the application in order to get visibility into the problem. These problems translate into three core logging challenges that will be reviewed in this section:

1. Cost: growing log volumes become expensive quickly

2. Developer productivity: writing, deploying, consuming and analyzing logs is a major distraction and time sink

3. Mean-time-to-resolve (MTTR): the friction in the logging process creates delays in fixing and repairing technical issues

In this chapter, we will explore each of these challenges in depth to more deeply understand the core challenges facing businesses when it comes to logging.

1. Cost

The volume of logs that companies are producing is spiraling out of control, and with it the associated costs.

Below are some of the key challenges that are behind this rise in costs.

Logging Sprawl

Developers tend to over-log their applications, as there is no feedback loop to prevent them from doing so.

Logs are a 'just-in-case' measure. They are often used to create the sense of safety - developers end up writing a log for every conceivable situation to ensure each incident or troubleshooting session will end in mitigation.

In addition, as logs can only be added during development and require a (relatively) long release cycle for every new augmentation, a tendency to log as much as one can during development naturally emerges.

Regardless of the actual problem in hand, the tool developers reach out to is often the same: add more logs, that results in sprawling and spiraling log volumes, without any easily-available alternative solution.

Logging Complexity

Cloud-native applications (which are growing in popularity) are more complex and have more constituent parts, resulting in greater log volumes.

As cloud-native software development becomes the norm, a large array of tools have been either heavily adapted or specifically written for cloud-native workloads: APMs, logging utilities, observability tooling, network monitoring tools, etc.

In addition, since every change made and deployed is adding complexity to an already complex set of systems, observing everything in unison (and each part individually) to ensure smooth operation is critical.

In cloud-native environments, developers are facing ever-increasing levels of complexity and find themselves relying on logs more than ever in order to get visibility into their applications.

Logging Imprecision

There is no way to know in advance which logs one will need, so developers create hotfixes, saturating application codebases with one-time, “garbage” logs.

Even with massive log coverage, it’s very common for a troubleshooting developer to not have all the information required at hand - mostly due to the concept of “unknown unknowns” - there’s no way of knowing in advance everything that might be needed, at any given point in time.

An interesting, repeating fact that we’ve learned through talking to our customer engineering teams, is that the vast majority of logs created are never consumed.

In fact, the practice of deploying a hotfix with more logs is so common that some companies write playbooks specifically for adding new telemetry in production. That information, naturally, generates a new set of questions to be answered, which sparks another cycle of hotfixes, and so the cycle repeats itself until the developer exhausts all the questions they needed to ask of the relevant part of the application.

Logging in advance is imprecise - causing developers to follow a “rinse-and-repeat” pattern of adding logging-only hotfixes to enrich the existing logging with more granular information, resulting in increased logging volumes.

Business Impact

Logging volumes are spiraling because of the complexity of the applications, the need to log for every conceivable problem and the imprecision of the original logs.

As the volumes spiral, so - naturally - does the cost.

We’ve seen among our customers that it is not unusual for companies that process large volumes of data or incur large traffic spikes to spend over $2M a year on logging ingestion and storage, alone.

Current logging practices leave engineering organizations with an expensive, imprecise and non-ergonomic experience.

2. Developer Productivity

Logs don't directly add any value to the software we build or to the customers we serve - they often fall under the category of "non-functional requirements": they help practitioners understand what's going on when things go wrong (or when they're simply hard to comprehend), which might otherwise cost time, money and reputation to fix.

The problem is that logging is not a frictionless process. It takes the developers away from value-adding activities (like writing code) and forces them to spend time and energy writing and deploying non-functional requirements. This costs them time - which is easily translated into monetary values - and reduces the overall time spent adding value to the business.

In this section we'll explore some of the main impacts that logging can have on developer productivity.

Deployment Times

The logging lifecycle is surprisingly long and complex, comprised of many moving parts, and prone to failure due to reasons that are not always up to the developer.

Application logs are usually added during development and consumed by engineers when the apps are running, in production and other "live" environments.

When the situation calls for more logs, a hotfix usually requires going through the whole CI/CD pipeline - even for a single log line. This costs time (however long the CI/CD pipeline takes to run), and can fail due to flaky tests or problematic configuration parameters (which are abstracted away from most developers and require a deep understanding of how the specific pipeline has been built).

In the fastest companies we've surveyed, a release takes 5-6 minutes. More commonly, though, we've seen pipelines that takes 30-50 minutes to run, and more often, multiple hours (and, in extreme cases, multiple days). Furthermore, pipelines fail due to many reasons are removed from developers - requiring a restart to the entire pipeline after it is fixed.

Context Switching

Context switching is a form of multitasking requiring a developer to move between unrelated tasks.

Every time a developer has to interrupt working on core business logic (i.e. switch context), it interrupts his flow, making it harder for him to resume working at the same pace and quality as before.

The fact that developers are continuously having to write new logs, wait for them to be deployed, analyze the logs and then finally fix the problem is a massive source of context switching.

In addition, the fact that logs are consumed outside of the developer's IDE requires them to jump out of what they are doing to attend to their logs, which constitutes another significant context switch.

Over time, having to frequently switch context reduces productivity, decreases creativity and results in lower quality software.

Hotfixing

Hotfixing involves releasing a relatively minor engineering update to a live (hence, "hot") system to resolve a bug or issue (or, more often than not, to add logs that were missing during troubleshooting).

If the current logs are not providing enough information on a given problem, there's a need to add new logs to get that extra bit of granularity.

Hotfixing is a quick way to rapidly deploy logs to a live system. However, instrumenting the application is not always a matter of just writing the logs - there's a need to write the logs, commit the to the source code repository, trigger the correct pipeline for that piece of code, ensure no side effects happen, reproduce the relevant state, then re-trigger the log-emitting action.Furthermore, developers tend to require several rounds of hotfixing.

In addition, it's often the case that the first round clarifies the situation but also raises new questions - which then require further investigation (and hence more hotfixes, and more setup time).

Developer & DevOps Friction

In the modern software world, there is often a split or barrier between the development and operations teams that creates a great deal of friction when communication and collaboration is required between them to understand what's going on in a live system.

Developers and operations are often unfamiliar with the processes, responsibilities, tooling and languages that the other side uses. This leads to miscommunications and delays when deploying logs requires working partly on the infrastructure (the domain of operations) and the application (the domain of the developers).

Business Impact

The sum impact of all these different challenges is that developers:

- Spend less time on value-add tasks
- Produce lower quality work as a result of context switching and distractions
- Are less happy because due to high friction and fragmentation in their workflows

At the business level this then translates into:

- Slower innovation
- Less satisfied customers
- Less satisfied employees

As developers are expensive to hire, it's especially important that they are working on the most valuable projects and not spending copious amounts of time on non-functional requirements.

The above business impacts all translate into missed revenue streams in terms of enhancements, overall system reliability and developer attrition.

3. Mean-Time-To-Resolve (MTTR)

One of the most important—but under-appreciated metrics in software development is mean-time-to-resolve (MTTR).

This is the average time it takes to fix a bug or failure, and return an application to its operational state, as well as - hopefully - ensuring that the failure won't happen again.

The reason why technical leadership teams are slowly starting to become more and more interested in this metric is that there is a strong correlation between MTTR and customer satisfaction, as traditionally quantified by NPS surveys.

If MTTR is poor, customer experience will suffer greatly and customers will start to churn. Conversely, if MTTR is excellent, customers will scarcely notice errors and outages and their experience will not suffer because of it.

The following section breaks down the various tenants that contributes to the increase of MTTR over time.

Deployment Time

When deploying logs to find the source of an issue takes a long time, the time it takes to resolve that issue is extended equally.

Often, when something goes wrong, the logs and other telemetry required to get down to the root cause of the issue is not in place, as we've explored above.

In this case, to begin resolving the problem there's a need to deploy new logs to get the information required for proper mitigation.

However, as mentioned before, deploying more telemetry requires going through the whole CI/CD cycle - and the longer the CI/CD cycle, the longer it takes to troubleshoot the issue at hand.

Hotfixing "Rounds"

When developers issue hotfixes to add more logs to a live application, there is a tendency to require several rounds of hotfixing, each of which causes more delays and friction.

When developers issue a hotfix with more logs, the information that the new logs reveal often raises more questions, which requires several more rounds of hotfixing to address, prior to the developer has full visibility of the situation.

Each round of hotfixing requires the developer to redeploy the application and do all the required setup for emitting the log. When multiplied by the number of hotfixes

Division of responsibility

Some issues that arise fall into the 'gap' between development and operations. It is unclear where responsibility lies, which generates frictions and delays the resolution of an issue.

When a bug is partly an infrastructure issue and partly an application issue, it is unclear whether the devs (who look after the application) or the ops (who look after the infrastructure) should take responsibility. This includes taking responsibility for instrumenting the logs that helps resolve the bug.

The above situation means, that, to deploy the right logs, the two sides end up having to spend a lot of time going back and forth, creating a lot of friction and delays. They are also less familiar with each others' toolsets, which exacerbates the friction.

Distance From the Issue

Best practice generally dictates that developers should not directly connect to production machines, which means that adding logs is the only way to learn about the true state of the application process or the machine that runs it.

The sensitivity of the customer data, the potential security risks, and the constant fear of making an error in production means that developers are often forbidden from connecting to machines running production workloads directly.

Furthermore, with the seemingly never-ending expansion of cloud computation resource usage, developers are not only physically distant from the machines that are running their code, they're also conceptually removed from the production computer, with most providers running virtual machines (or derivatives thereof) and not physical hardware, developers can't interact with as they used to.

When developers can't connect directly to get a sense of the state of the running application, they often resort to the next best things - logs and telemetry - which, when factoring in the current workflows around telemetry, provide a poor replacement for a direct connection.

Business Impact

The impact of poor MTTR affects a few areas of the business.

On one hand, it delays the resolution of technical issues that often takes up an unnecessary amount of time, and, negatively impacting developer productivity, and slowing down innovation.

On the other hand, it results in overly-long delays to resolve technical issues, and negatively affects the customer experience. The hold-ups that issues related to logging cause in terms of delaying the resolution of technical issues results in degradation of service to the customer. It can even result in companies not being able to meet their SLAs.

MTTR is now a major indicator of the technical health of engineering organizations, with poor performance resulting in less productivity, slower innovation and a poorer customer experience.

The Lightrun Developer Observability Platform

Lightrun is a Developer Observability Platform that enables developers to add logs, metrics and traces to live applications in real-time, on-demand. The outputs are then available for consumption immediately from the IDE, CLI or APM of choice.

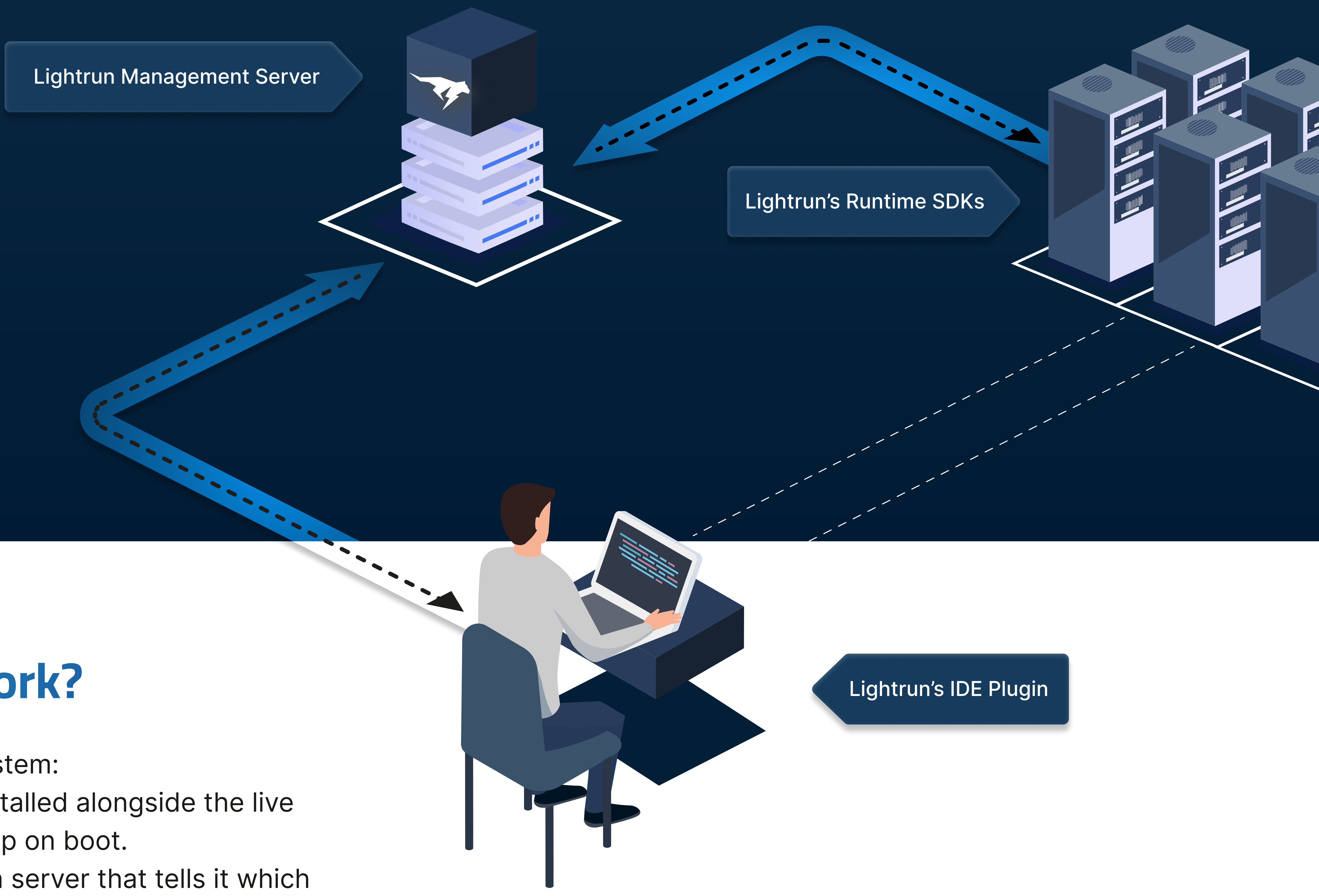
With Lightrun's revolutionary approach to observability, the platform enable engineering teams to connect to their live applications and continuously identify critical issues without continuous hotfixes, redeployments or restarts.

The Lightrun Sandbox™ ensures there are no side effects or unexpected performance overhead to the live application, and verified the integrity, stability and security of the instrumentation. Lightrun meets the developers where they are: in the cloud, in micro-services and in serverless functions, in bare-metal servers and Kubernetes - Lightrun is platform-agnostic and relies on the runtime environment alone in order to run.

FinOps FOUNDATION General Partner

Gartner COOL VENDOR 2021

In addition, Lightrun was recently named a Gartner Cool Vendor in Observability and Cloud Operations, and is a member of the FinOps Foundation.



How Does It Work?

Lightrun is a three-tier system:

1. An runtime agent is installed alongside the live application and spins up on boot.
2. It communicates with a server that tells it which logs to add and where to add them.
3. A suite of IDE plugins / our CLI client talks to the server, allowing developers to indirectly "connect" to their live applications, no SSH or legacy remote debugging protocols required.

The agent adds the new code for the new logs on-the-fly using high-end instrumentation techniques for our supported runtimes. No actual source code ever leaves the premises and no source code is modified on in the source control system (i.e. GitHub, GitLab, etc..) .

When the app executes that code, those added logs are emitted just like normal application logs and can be sent to a console, a file, an external observability system, or back to the server and then onto the client (usually an IDE plugin or a CLI client).

This setup allows for the addition of new log statements into a running application at any point, and removes the need to reproduce specific state or conditions during troubleshooting.

Dynamic Logging With Lightrun

Lightrun’s platform allows engineering teams to fundamentally change their approach to instrumenting telemetry by introducing dynamic logging.

The best way, by far, to reduce logging costs is - unsurprisingly - to log less. A great way to do so is to use Lightrun to replace a portion of the codebase’s **static** logs (i.e. logs added during development, which means, all the logs in most production systems) with dynamic logs (i.e. logs added only during the runtime of the application).

Dynamic logs are those that can be added to an application at runtime without requiring source code modification or triggering a whole new development cycle.

Dynamic logs are:

Ephemeral
Lightrun logs “live” and “die” as required. They are added at runtime, do not persist in the codebase, and are only emitted so long as they are ‘live’.

Real-time
Lightrun logs can be added immediately to live applications, i.e. they can be instrumented and consumed on the fly, in real time, without having to redeploy the application.

Conditional
Lightrun logs can be emitted with very precise conditionality. For example, they can be emitted for one machine or an entire production fleet. Similarly, developers can choose to log for a specific user or class of users, e.g. people from a specific geography.

IDE-First
Lightrun logs can be instrumented and consumed right from the IDE. This increases developer productivity and velocity by reducing the need for context switching while troubleshooting.

Optionally Intersperseable
If need be, Lightrun logs can be timestamped and correlated with existing application logs, and consumed by the same observability systems.

Granular
Lightrun logs provide the exact information needed, at the time it is needed (instead of logging everything possible, then “mining” (i.e. analyzing) the information for the exact bit required).

Using Lightrun allows developers to drastically reduce emitted log volumes by elmininating logs that were used to “cover” every possible eventuality.

Instead of relying on these logs, developers can add dynamic Lightrun logs on the fly to instantly get visibility into the issue at hand, without needing to deploy a hotfix or redeploy the application.

Benefits of Dynamic Logging with Lightrun

Technical Benefits

Slash log Volumes

Using dynamic logging, developers can write highly-targeted logs that pinpoint telemetry to specific users/machines. These logs are also ephemeral - meaning they are automatically removed once they are no longer required (often after a set, pre-configured period of time).

This means that instead of logging everything and analyzing later, only critical logs will be added to the code in development. If there's a new metric to add or a specific aspect of the application that requires additional logging, the dynamic logging approach allows developers to instrument those pieces of telemetry on the fly - without having to redeploy the application.

The result is that the majority of logs previously required can now be eliminated, allowing developers to focus solely on the logs needed for compliance and normal operation of the application - while adding more logs on the fly, as the need arises.

Lightrun, on average, enables engineering organizations to reduce logging volumes by 60%.

Eliminate the Need to Redeploy the Application

The process of dynamic logging is shorter, simpler and more ergonomic than the traditional, static logging workflow.

Instrumenting a static log consists of many different steps - mainly due to the need to go through the entire CI/CD pipeline, which can be lengthy and prone to test failures (related or unrelated to the change at hand).

Lightrun, however, is based on real-time instrumentation that does not rely on actual code being introduced into the codebase. This approach - called dynamic logging - is secure, highly-performant, and read-only while eliminating the need to redeploy the application. This makes the overall process much simpler and shorter.

Reduce Context Switching

Because logs can be instrumented and consumed from the IDE, developers don't have to switch contexts when troubleshooting between their development environment, the CI/CD pipeline tooling, and their APM or logging platform.

This helps the developers to stay focused on the job at hand, eliminates distractions and delays, and generally reduces the friction involved with shedding more light on the live execution of the application.

Reduce Irrelevant Noise and Increase Granularity

When logging everything, an issue rising up to the surface requires reviewing, analyzing and extracting valuable information from a large volume of data.

The high granularity of dynamic logs allows developers to choose what to log and when to log it, instead of logging everything and “mining” the information later for valuable data.

Reduce Mean-Time-To-Resolve (MTTR)

Troubleshooting complex systems often involves the addition of more telemetry (via hotfixes, which requires redeployments), the consumption of said telemetry, and the addition of more telemetry following what the developer learned from the previous iterations.

This is a non-agile, iterative process that increases the time it take to resolve incidents (often measured in terms of MTTR) due to the long CI/CD cycles required for a large amount of hotfixes.

Lightrun, instead, opts to add logs dynamically, at runtime, and route the information directly to the IDE. By doing so, it enables developers to drastically reduce debugging time and allows for a more ergonomic troubleshooting experience.

Reduce Team Friction

Lightrun logs can be added dynamically throughout the SDLC: in dev, in QA, in staging, in CI, and even in production. Regardless of the environment, Lightrun logs are added and consumed just the same - in real time, on demand and right into the developer’s IDE or CLI.

In practice, that means that developers are empowered to ask questions and get answers themselves, rather than being required to consult with other team members (most notably DevOps engineers, who operate the infrastructure running the application) to get visibility into application’s state. This helps reduce the dependency on - and by proxy the friction with - the developer’s team members.

Business Benefits

The technical benefits above translate directly into significant business benefits:

Massively Lower Costs

When lowering the volume of static logs emitted and enabling developers to get an even improved level of clarity using dynamic logs, engineering organizations reduce the overall costs of logging and observability while maintaining the same ergonomic developer experience.

See Chapter 3 (*Lightrun’s Economic Impact on Enterprise Logging & Observability Costs*) for a breakdown of the cost savings that can be achieved with dynamic logging.

Lightrun, on average, enables engineering organizations to reduce logging costs by 30%.

Reduce MTTR & Decrease Time to Market

Logging is a non-functional requirement, not a customer-oriented value-add. The less time and energy developers spend writing and maintaining application logs, the more time they have to focus on building software that customers want to use.

By reducing context switching, removing the need to redeploy in each addition of telemetry, and improving the troubleshooting experience, dynamic logging can significantly accelerate the entire software development lifecycle.

Improve Developer Productivity and Experience

Dynamic logs create a more ergonomic experience of troubleshooting, with less reliance on operational requirements - like redeploying and writing queries inside the APM / logging platform.

In addition, Lightrun works completely within the development environment - every Lightrun action can be instrumented and consumed from the same interface the application code is written in.

This reduction in friction results in more productive developers, spending less time on configuration and operational work, and more time writing new features.

Reducing Logging Costs with Dynamic Logging in Practice

This section comes as a reference to a few, select external sources that explain - in more depth - how to perform some groundwork prior to implementing dynamic logging in your organisation, in order to extract the most value out of the practice.

There are three main steps to the process: first, an organisation must **perform a log audit** to take inventory of the existing logging sitaution. Then, all **reproducible, static logs must be removed** as they can easily be replaced with dynamic logs. Finally, an **optimization of the remaining logs** should take place in order to cement the benefits.

Performing A Log Audit

A log audit includes a review of the logging output of the entire system, picking apart the specific portions that produce the logs that cost the most, and highlighting which of these can be considered for removal. This amounts to "taking inventory" of the system, and enables the organisation to get an initial estimate of how valuable dynamic logging can potentially be once implemented.

Replacing Reproducible Logs

Reproducible logs are logs that can be easily re-emitted by developers on local machines or on staging environments - in other words, . As the cost for recreating them is low, they're great candidates for removal and re-instrumentation using dynamic logging in production. The associated risk is low and the potential benefit is high - saving on both precious developer time and observability costs.

Optimize the Remaining Logs

It's advisable to follow the Pareto principle (also known as the 80/20 rule) in any optimization effort. In the case of logging, it's beast to focus on high-frequency, high-cost logs. The way to do it in the context of the rest of the codebase is to find 'big offender' logs - that are specific, local maxima points of log emissions - as well as identify other opportunities for global optimizations in every application module.

Lightrun's Economic Impact

Understanding the economic impact of any system on an organization, needs to cover not only the primary cost savings - how much reduction was incurred in license fees and/or compute resources - but also secondary savings that are harder to quantify (like developer hours reclaimed or how much faster the support process becomes).

The following chapter explores the impact of Lightrun from three perspectives:

- **Costs**
- **Developer Productivity**
- **MTTR (Mean Time To Resolve)**

While covering each perspective, we'll look at 3 different metrics to convey the impact Lightrun has on the entire engineering organization: cost savings incurred after implementing the system, 3-year ROI of the implementation & the payback period for using Lightrun in the organization.

1. Cost

TOTAL COST OF LOGGING WITHOUT LIGHTRUN

Dynamic logging has a major effect on the logging costs incurred by engineering organization.

To keep things simple yet offer a robust framework of exploration, we've chosen to review the cost factor via the lens of a common use case: a large engineering team, working with a managed, centralized observability vendor.

We'll use as an example, an application with a relatively high amount of transactions, and look at the logging volumes they bring to the table:

- 40TB of monthly ingested logs
- 9B Monthly log events (see note below)
- 30-day retention

Note: A "log event" is a metric used to denote processing / analysis of logs in order to have them available for querying. These are reflected differently on the pricing of each observability system, and so we've grouped them under their term "log events".

The following table breaks apart the cost of logging in said application, considering the logs are instrumented **statically, during development** (as the current practice goes) and analyzed using a centralized, managed observability system in a production setting:

LOGGING COSTS, PRE-LIGHTRUN				
REF.	METRIC	CALC.	FIGURES	NOTES
A1	# of Ingested Logs (Monthly)	Customer Case Study	40TB	
A2	# of Log Events (Monthly)	Customer Case Study	9B	
A3	# of Retention Days	Customer Case Study	30	
A4	Yearly Log Transmission / Egress Costs	$12*A1*0.09	$43,200	Based on AWS data egress pricing
A5	Yearly Log Ingestion Costs	$12*A1*0.1	$48,000	
A6	Yearly Log Processing / Indexing Costs	$12*A2*3.75	$405,000	
A7	**Total Yearly Managed Observability Costs**	**A4+A5+A6**	**$496,200**	

Note: The figures above focus on logging alone, without focusing on the cost of the rest of the tooling often provided by observability vendors (who often provide additional tooling for various levels of the infrastructure aside from the code level).

TOTAL COST OF LOGGING WITH LIGHTRUN

Lightrun helps save on costs by:

- Swapping redundant static logs with as-required dynamic logging
- Reducing reliance on self-hosted or managed logging systems

Note: the table below factors in a 60% reduction of log volumes using Lightrun, following the case study and our own internal benchmarking.

LOGGING COSTS, POST-LIGHTRUN				
REF.	METRIC	CALC.	High Application Transaction Volume	NOTES
B1	# of Ingested Logs (Monthly)	A1*40%	16TB	Up to 60% log volume reduction
B2	# of Log Events (Monthly)	A2*40%	2.4B	Up to 60% log volume reduction
B3	# of Retention Days	A3	30	
B4	Yearly Log Transmission / Egress Costs	$12*B1*0.09	$17,280	Based on AWS data egress pricing
B5	Yearly Log Ingestion Costs	$12*B1*0.1	$19,200	
B6	Yearly Log Processing / Indexing Costs	$12*B2*3.75	$162,000	
B7	**Total Costs (Post-Lightrun)**	**B4+B5+B6**	**$198,480**	
C1	Lightrun Costs (Yearly)	Lightrun pricing	$162,000	
C2	Risk Adjustment	↑7%	$120,000	
D1	Total Costs (Pre-Lightrun)	A7	$496,200	
D2	Total Costs (Post-Lightrun)	B7+C1*(1+C2)	$340,774	
D3	**Total Cost Savings (Post-Lightrun)**	**D1-D2**	**$155,426**	
D4	**Total Cost Savings (Post-Lightrun) %**	**100%-(D2/D1)**	**31%**	

SUMMARY

31% reduction in logging costs

248.1% ROI in a 3-year period

< 5 Months to pay back the cost of the system

Total Cost Savings With Lightrun

Total Cost Savings (pre vs post Lightrun)	$297,720
Lightrun Costs	$120,000
Total Cost Savings Post-Lightrun	**$155,426**
Total Cost Savings Post-Lightrun (%)	**31%**

Three-year ROI

Three-year total cost savings (without Lightrun)	$893,160
Three-year Lightrun costs	$360,000
Three-year ROI (%)	**248.1%**

Payback Period

Total cost savings (pre vs post Lightrun)	$297,720
Lightrun costs	$120,000
Payback period	**< 5 Months**

In the long run, an organization that uses dynamic logging can save 100s of thousands to millions of dollars in logging costs simply by removing reproducible logs and replacing them with dynamic logs.

2. Developer Productivity

Dynamic logging also has a very noticeable effect on developer productivity.

By streamlining the practice of hotfixing and making it easier for developers to get real-time, on-demand information without the operational overhead, Lightrun saves on many small tasks often carried out by developers throughout their work day. These savings compound, and can be easily quantified (in developer hours) - resulting in significant cost savings over time.

The table below shows the accumulated benefits of using dynamic logging over a period of three years with the same scenario discussed in part one (large engineering team, application with a high transaction volume), and covers the following developer productivity metrics:

- Instrumenting metrics for performance testing
- Context switching
- Reduced hotfixing
- Enhanced focus / reduced friction
- Reduced profiling/APM usage

DEVELOPER PRODUCTIVITY OVER 3 YEARS

REF.	METRIC	CALC.	YEAR 1	YEAR 2	YEAR 3
A1	Number of developers		1000	1000	1000
A2	Percentage of developers using Lightrun		20%	30%	50%
A3	Number of developers using Lightrun	A1*A2	200	300	500
A4	Percentage of developer time spent Troubleshooting, Understanding Code and Exploring Codebases		30%	30%	30%
A5	Productivity Improvement: instrumenting metrics for performance testing		2%	2%	2%
A6	Productivity improvement: reduced context switching		3%	3%	3%
A7	Productivity improvement: reduced hotfixing		4%	4%	4%
A8	Productivity improvement: enhanced focus / reduced friction		3%	3%	3%
A9	Productivity Improvement: reduced profiling / APM usage		4%	4%	4%
A10	Productivity improvement from reduced technical debt	Technical debt is reduced over time	0%	2%	5%
A11	Total percentage improvement in development productivity	A5+A6+A7 +A8+A9+A10	16%	18%	21%
A12	Hours saved per developer, per year	2,080*A11*A4	100	112	131
A13	Total hours saved by developers	A3*A12	19968	33696	65520
A14	Percentage of time recaptured for productivity		50%	50%	50%
A15	**Hours recaptured for productivity**	**A13*A14**	**9984**	**16848**	**32760**
A16	Fully burdened hourly salary	$130K annually	$65	$65	$65
B1	Value of development recaptured	A15*A16	$648,960	$1,095,120	$2,129,400
B2	Risk adjustment	↓15%			
B3	**Value of development recaptured (risk-adjusted)**		**$551,616**	**$930,852**	**$1,809,990**

SUMMARY

21% improvement in developer productivity

1434.6% ROI over a 3-year period

119,184 developer hours saved over a 3-year period

< 2 months to pay back the cost of the system

Note: While not factored into the figures above, one should also factor in the revenue generated from new products and features developers would be able to create with these additional 59,592 hours of work every year.

Three-year ROI

Three-Year Development Spending Reclaimed for Value-Add Work	$3,873,480
Three-Year Lightrun Costs	$270,000
Three-Year ROI	**1434.6%**

Payback Period

Total Year-One Development Spending Recaptured	$648,960
Lightrun Costs	$90,000
Payback Period	**< 2 Months**

Both the ROI and the time-to-pay-for-itself will accelerate over time as more and more developers start using the platform and get more accustomed to this way of working.

Using dynamic logging with Lightrun can increase productivity and save developers 10s of thousands of hours in aggregate.

3. MTTR (Mean Time To Resolve)

In this section, we'll explore how dynamic logging impacts the average time to resolve incidents in production, usually referenced in terms of MTTR.

Dynamic logging affects this process by aiding developers in the following ways:

- Eliminating the need for application redeployment
- Eliminating the need for debug-log hotfixing (i.e. hotfixing just to add more telemetry)
- Preventing repetitive, non-productive developer work
- Healing the division of responsibility between Dev and Ops
- Allowing developers to directly find out what is going on in a machine in real time

The table below shows the accumulated benefits of using dynamic logging over a period of three years, covering the following MTTR metrics:

- Reliance on properly-functioning production systems
- Revenue at risk due to product failures and downtime
- Revenue recaptured by improved uptime and performance

MTTR OVER 3 YEARS

REF.	METRIC	CALC.	YEAR 1	YEAR 2	YEAR 3
A1	RR - Annual Recurring Revenue	Industry Scaleup Composite	$5,000,000	$12,000,000	$20,000,000
A2	Percentage of revenue that relies on properly-functioning production systems	Industry Scaleup Composite	80%	80%	80%
A3	Percentage of revenue at risk for cancellation due to product failures and downtime	Industry Scaleup Composite	15%	15%	15%
A4	Total revenue at risk for cancellation due to product failures and downtime	A1*A2*A3	$600,000	$1,440,000	$2,400,000
A5	Revenue re-captured by better uptime and performance due to decreased MTTR	Internal Customer Information	30%	35%	40%
A5	Increased incremental revenue	A4*A5	$180,000	$504,000	$960,000
B1	Risk Adjustment	↓12%			
B2	**Increased incremental revenue (risk-adjusted)**		**$158,400**	**$443,520**	**$844,800**

SUMMARY

35% reduction in MTTR, on average

609% ROI over a 3-year period

< 7 months to pay back the cost of the system

Three-year ROI

Three-Year Increase in Incremental Revenue	$1,644,000
Three-Year Lightrun Costs	$270,000
Three-Year ROI	**609%**

Payback Period

Total Year-One Increase Incremental Revenue	$180,000
Lightrun Costs	$90,000
Payback Period	**< 7 Months**

Both the ROI and time-to-pay-for-itself will accelerate over time as more and more developers start using the platform over time and get more accustomed to this way of working.

CONCLUSION

Lightrun's Economic Impact Over a 3-Year Period

Observability Vendor Cost Savings	$466,278
Value of Developer Productivity Recaptured	$3,873,480
Value of Revenue Recaptured due to Reduced MTTR	$1,644,000
Lightrun Costs	-$360,000
Total Cost Savings	**$5,623,758**

CASE STUDY

Taboola:

Reducing MTTR & Saving 260+ Debugging Hours A Month

The Challenge

Taboola's production environment is particularly dynamic. With many different features in development simultaneously to accommodate the needs of the business, their developers push a large number of changes on a daily basis.

Taboola was looking for a solution that will enable them to make sure each released version works without a hitch. Ultimately, they looked for a tool that would allow them to troubleshoot issues and validate feature behavior in production services in a quick, developer-friendly

The Result

With instant, real-time production logs, snapshots, and metrics, Taboola developers now save precious incident resolution time previously spent waiting for their hotfixes to deploy. Using Lightrun on a constant basis decreases MTTR, increases the rate at which Taboola deploys new features to production, and improves each individual developer's productivity.

The process has resulted in over 260 debugging hours saved every month.

Read the full case study here>

CASE STUDY

Start.Io:

Reducing MTTR By 50-60%

The Challenge

Start.io handles more than 30 billion requests a day, resulting in complicated production issues. Most notably, concurrency, parallelism issues that are hard to replicate locally and can lead to severe outages, as well as problems with software-defined caches.

The Result

By leveraging Lightrun, the client' developers could write conditional logs that sends them proactive alerts when the issues in question occur. This "catches" the issue and sends the report straight into the developer's IDE.

This process has accelerated the client incident resolution by 50-60%.

Read the full case study here>

Final Thoughts

Organizations are forced to log more than they should because of a (observed) lack of alternatives. As a result, costs spiral, developer productivity is negatively impacted, and MTTR is on a constant rise - especially in complex, cloud-native environments.

By using dynamic logging with Lightrun developers can fundamentally change how their organization approaches logging. Instead of “logging everything and analyzing later”, developers can now log on an “as-required” basis - log only “what you need, when you need it”.

This can cut log volumes (and the associated costs) by up to 40%, improve developer productivity significantly, reduce mean-time-to-resolve, improve time to market, and create higher-quality software.

Find this document interesting and would like to implement our playbook in your organisation?

Book a meeting with us>