


The state of Observability and Security of Cloud Native Software

Presented by- Niolet D'mello



\$ cat about_me

- Product & Application Security Engineer @ Datadog
- Previously:
 - Platforms & Infrastructure engineering @ Intel
 - Software engineer @ McAfee
 - Lecturer @ Polytechnic College
- Bachelor in Computer Engg/ Master of Science- Software Engg
- Passionate about Writing
- STEM mentor @ SJSU
- Speaker @ national and international Cybersecurity conferences

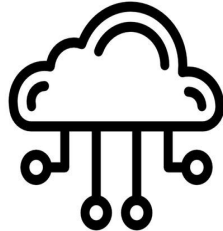


Legal Disclaimer

“All views, thoughts, and opinions expressed in the presentation belong solely to the speaker, and not necessarily to their employers, organizations, committees or other groups or individuals. ”



What is Cloud Native Software?



Characteristics & Design Considerations

- Cloud native applications are applications that are distributed in nature and utilize cloud infrastructure
- In designing cloud native applications we consider five key areas when starting with the initial design:

Operational
excellence

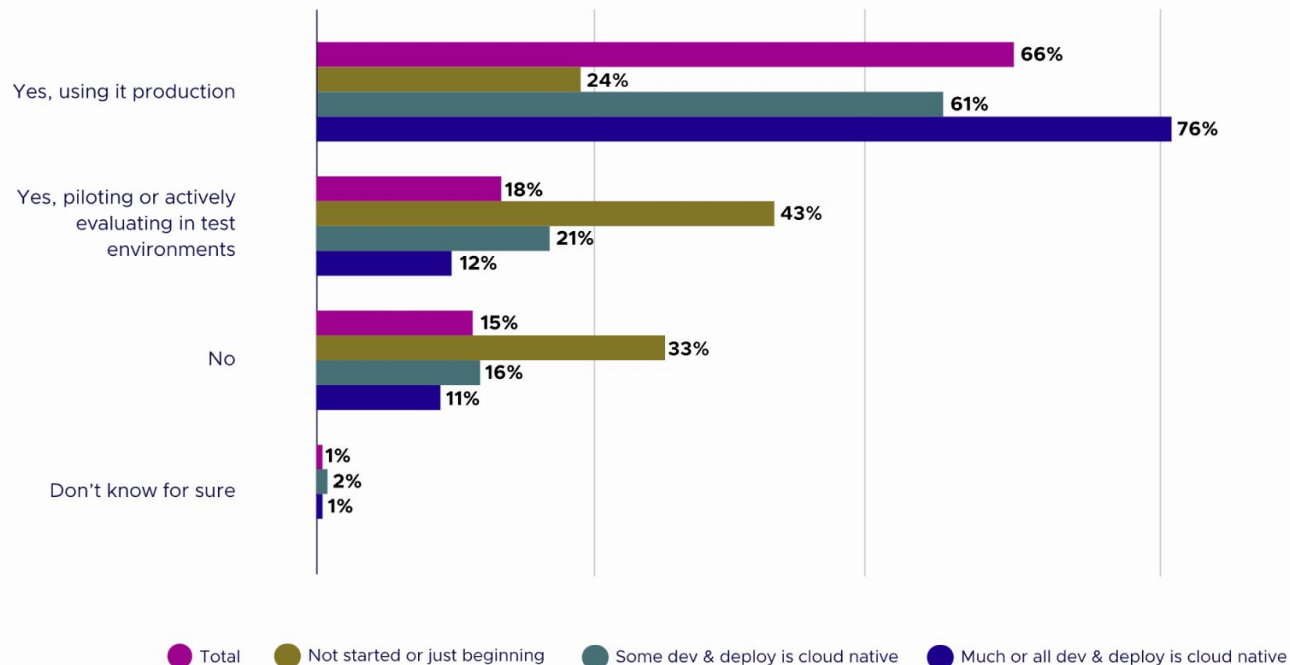
Security

Reliability

Scalability

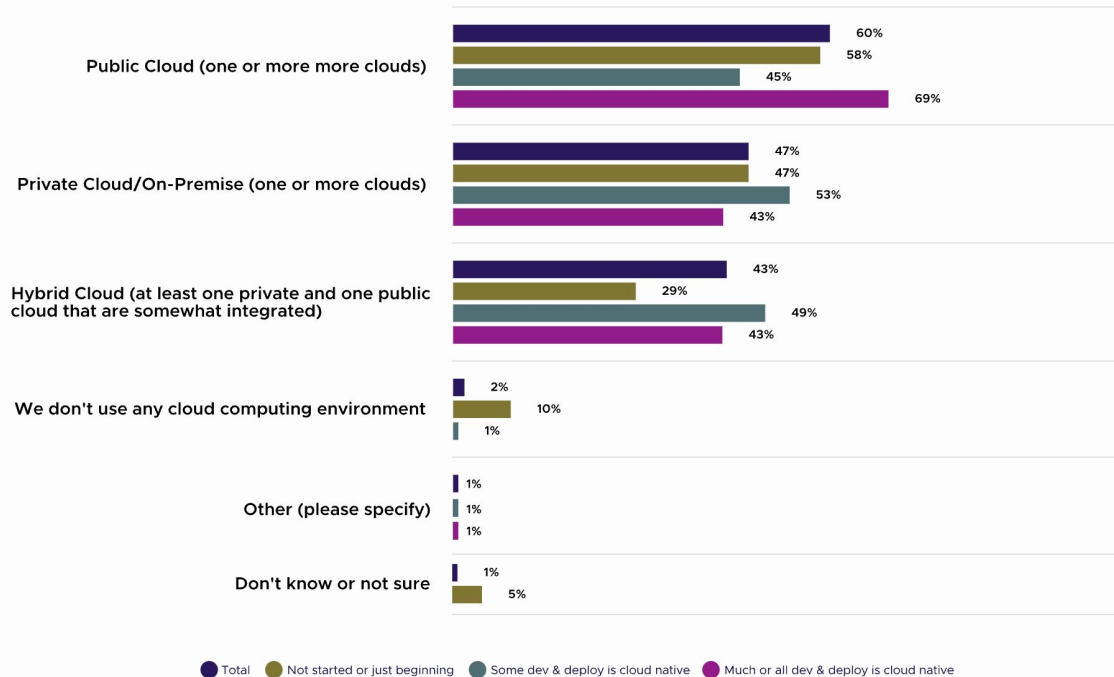
Cost

To What Extent Has Your Organization Adopted Cloud Native Technologies?

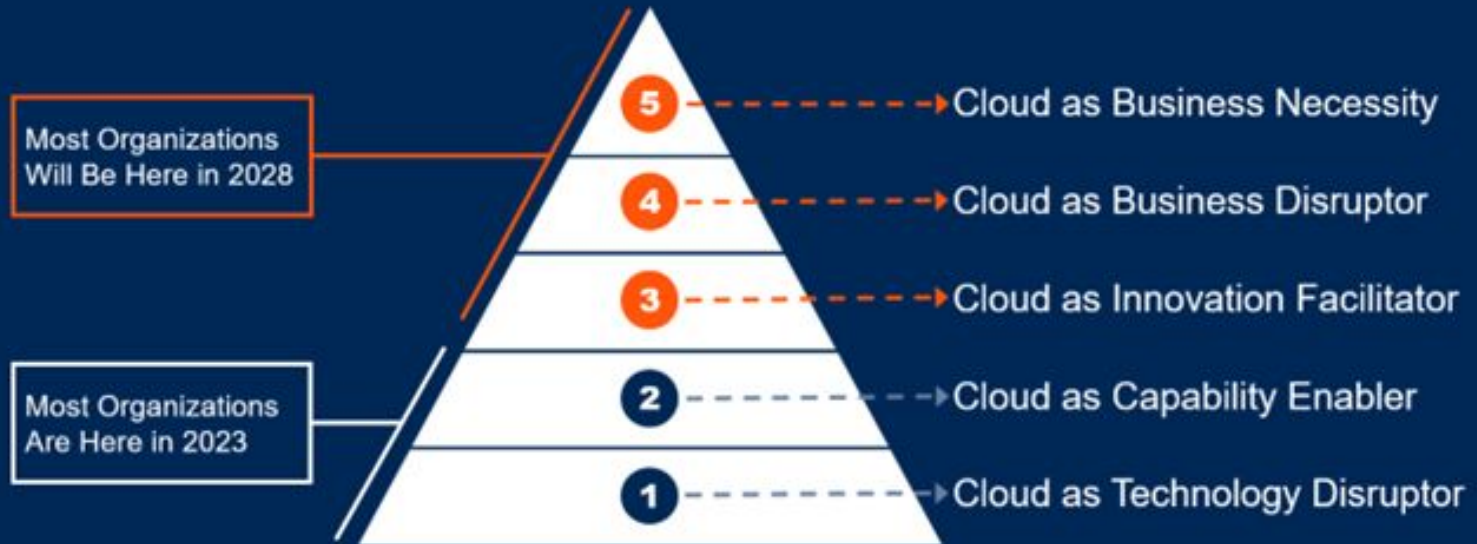


Which Of The Following Combinations Of Data Center And Cloud Architectures Does Your Organization Use?

Segmented By Adoption

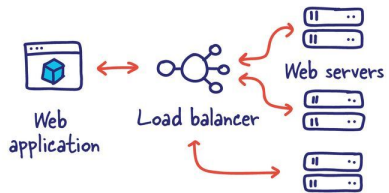


Cloud in 2028: From Technology Disruptor to Business Necessity



8 FALLACIES & DISTRIBUTED SYSTEMS

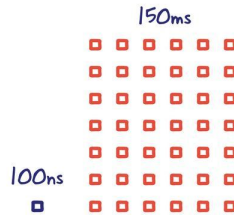
1 The network is reliable.



6 There is one administrator.



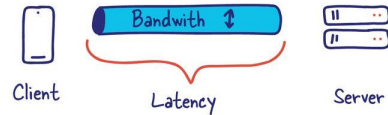
2 Latency is zero.



7 Transport cost is zero.



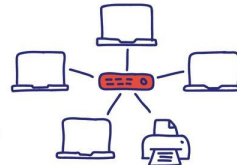
3 Bandwidth is infinite.



4 The network is secure.



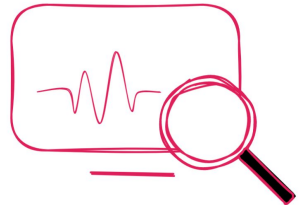
5 Topology doesn't change.



8 The network is homogeneous.



What is Observability?



Characteristics & Design Considerations

- Observability has become an integral part of how many organizations build and maintain their systems, helping to improve performance, reduce toil, and save money through better resource utilization.
- It provides the ability to understand and gain insights into the internal state of a system based on its external outputs.

Why

- Modern software development challenges drive the need for robust observability
- Several factors in how we build and operate software today:
 - Time to get code out to Prod \sim fast feedback loops
 - Shipping, release, deployment styles
 - Platform teams, ownership of services
- Resource Efficiency
- Capacity Planning
- Improved User Experience
- Cost control

Importance in Cloud Native Environments

- Microservices architectures introduce more communication points and potential failures
- Container orchestration adds layers of abstraction
- Dynamic scaling means the system is constantly changing
- Infrastructure as code and CI/CD lead to frequent changes that need to be tracked

The Monitoring vs. Observability Paradigm

- Focuses on understanding "unknown unknowns" - issues you didn't anticipate
- Provides context for troubleshooting rather than just alerting that something is wrong
- Supports complex analysis and correlation across multiple dimensions and services
- Enables engineers to ask new questions about system behavior without deploying new instrumentation

The 3 Pillars of Observability

Logs

- Textual records of events in a system
- Provide the most granular level of detail (e.g., error messages, warnings)
- Useful for post-incident investigations, debugging, and for capturing contextual information around system events

Metrics

- Numerical measures that quantify system behavior (e.g., CPU usage, memory consumption, request counts, error rates)
- Enable monitoring trends over time and setting up thresholds for alerting
- Lightweight to store and easy to aggregate for dashboards and quick insights

Traces

- End-to-end records of requests as they traverse across distributed services.
- Capture each microservice call, timing, and context within a single “trace ID”
- Essential for diagnosing latency issues or identifying which service(s) is slowing down an entire request path

Observability is hard



Because software is hard..Increased complexity



Ephemeral infrastructure..Not just the code you write but the resources too



Data volume challenges



Signal-to-noise ratio



Consistency issues

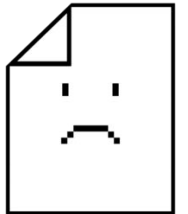


Tool proliferation



Talent and skills gap

Observability Pain Points & Examples



Distributed Context Loss

- Pain Point: Tracking a request across multiple services and understanding its complete journey.
- Example: A ride sharing application where-
 - A user requests a ride via a mobile app. The Dispatch Service logs an error indicating “No available drivers.” The actual issue is the Location Service timing out while verifying the user’s GPS data. Because the Dispatch Service never receives a valid location, it fails to match the user with a driver.
 - Without distributed tracing to connect these services, engineers might spend hours investigating the Dispatch Service error logs, unaware the real bottleneck is a slow or failing Location Service call.
 - Tracing would reveal that the ride request reached Dispatch successfully but failed downstream when the location data wasn’t received in time.

Unknown Dependencies

- Pain Point: Hidden or undocumented dependencies between services cause unexpected impacts.
- Example: A DevOps team reduces CPU allocation for a seemingly low-priority caching service. This unknowingly affects the authentication service that depends on it, causing intermittent login failures that appear unrelated to the original change.

Alert Fatigue & Noise

- Pain Point: Too many alerts, many false positives, leading to important signals being missed.
- Example: An operations team receives over 200 alerts daily from their monitoring systems. When a critical database begins experiencing slow queries, the alert is lost among numerous low-priority notifications, delaying response by several hours.

Inconsistent Instrumentation

- Pain Point: Different teams implement observability differently, making correlation difficult.
- Example: The payments team logs in JSON format with detailed error codes, while the user authentication team uses plain text logs with different terminology for similar events. During an incident, engineers struggle to piece together what happened when a user payment fails after authentication.

Cost and Resource Management

- Pain Point: Storing and processing vast amounts of observability data becomes expensive.
- Example: A company implements full tracing across all services but finds their cloud storage costs increase by \$20,000 monthly due to the volume of data. They're forced to reduce sampling rates, potentially missing important traces during critical incidents.

Cost, Scalability, and Tooling Trade - Offs



Balancing High Fidelity vs. Budget

- Collecting 100% of logs, metrics, or traces can be costly.
- Evaluate sampling strategies (e.g., probabilistic tracing) or tiered data retention to control expenses.

Scale Considerations

- Explosive data growth from microservices requires horizontally scalable observability platforms.
- Containerized workloads can dynamically scale up or down, leading to spiky telemetry data volume.

Choosing Tools & Platforms

- Open Source (Prometheus, Jaeger, Elasticsearch) vs. Managed Services (Datadog, New Relic, Splunk).
- Trade convenience and advanced features for higher recurring costs in managed solutions.
- Evaluate cost factors like ingestion rates, data retention windows, and query performance.



Security aspects of cloud native software



Security in Cloud Native Applications

- Requires a fundamentally different approach than traditional security models
- Distributed architecture means larger attack surface
- Ephemeral resources create challenges for traditional security scanning
- Infrastructure as code introduces new security considerations
- API-centric design requires robust authentication and authorization
- Container and orchestration security becomes critical

Cloud Native Security Challenges

Expanded Attack
Surface

Credentials/Secrets
Management

Data in Transit

Kubernetes
Security

Configuration
Drift

Ephemeral
Resources

Container
Security

CI/CD
Security



Convergence of Observability and Security



Cloud Native Security Observability

- Observability data serves dual purposes:
 - Operational reliability
 - Security monitoring and detection
- Log aggregation enables security analytics
- Tracing can identify unusual access patterns
- Metrics can detect abnormal resource usage indicating compromise
- Shared tooling reduces operational overhead
- Unified context for both reliability and security incidents



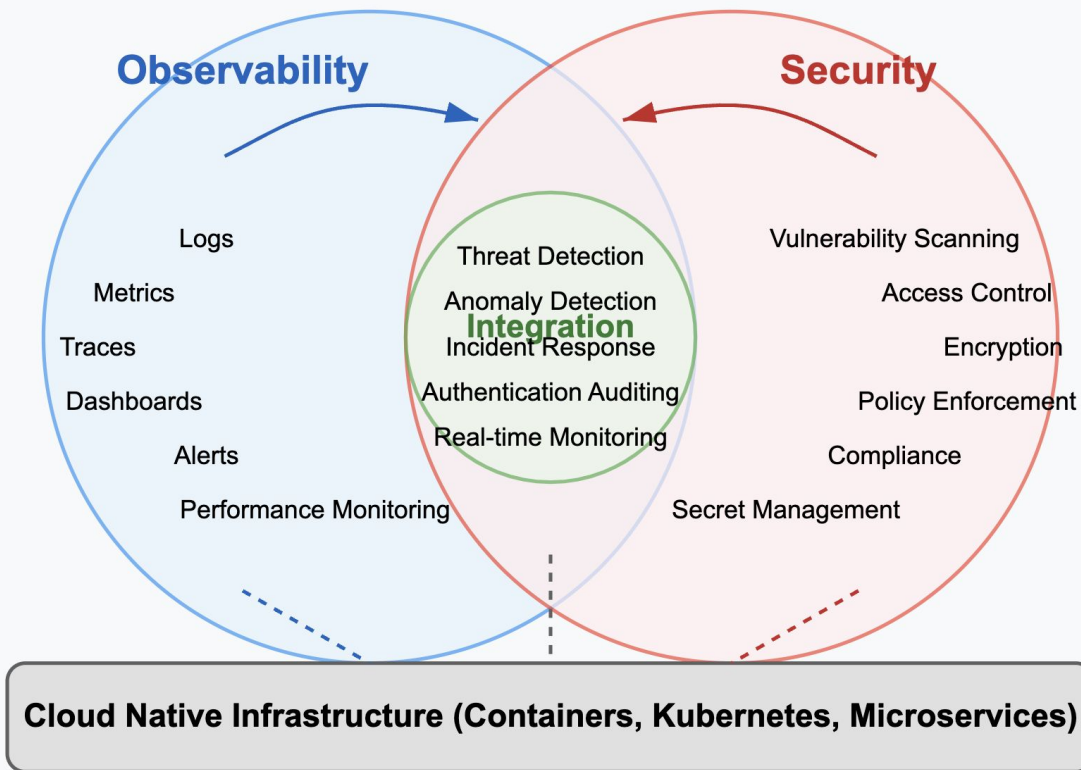
Security Observability Signals



Security Observability

- Authentication Anomalies: Failed login attempts, unusual access patterns
- Authorization Failures: Access denied events, privilege escalation attempts
- Infrastructure Changes: Unexpected modifications to the environment
- Network Traffic: Unusual communication patterns between services
- Resource Utilization: Unexpected spikes in CPU/memory (potential cryptomining)
- Process Behavior: Execution of unusual commands or binaries
- Data Access Patterns: Unusual data retrieval operations

Cloud Native Observability and Security Integration



Implementing Cloud Native Observability

Phase 1: Planning

Define goals, identify key services, select tools, establish standards



Phase 2: Implementation

Instrument code, configure collection, establish baselines



Phase 3: Integration

Connect with security tools, automation, CI/CD, alerts



Phase 4: Optimization

Refine data collection, improve alerts, cost management



Continuous Improvement

Log Analysis Challenge



Discussion Time



Q&A

Datadog Student Pack

- You can sign up for the Github Student Pack here:
<https://education.github.com/pack>
- And once the Github account is enrolled in that program, you can go here to create a Datadog student account:
<https://studentpack.datadoghq.com/>



DATADOG