Rust in the Cloud: Building AWS Applications That Scale

Conf42 Rust 2025

Vamsi Praveen K

Why Rust for AWS?

Performance at scale

Memory safety guarantees

Small, efficient binaries

Lower cloud costs

Rust vs Other Languages for AWS

Quick comparison focus: Lambda, containers, I/O, ecosystem, developer speed

Language	Strengths	Limitations
Go	fast startup, simple concurrency, smaller runtime	runtime bigger than Rust
Java	mature ecosystem, rich AWS libs	heavier cold starts
Node.js	quick dev, rich packages	single-threaded CPU limits
Python	fastest dev, tons of libs	slower runtime for CPU tasks
C++	peak perf, manual memory	higher complexity
Rust	performance plus safety, predictable latency, small binaries	steeper learning curve

Where Rust Wins Today

Optimized Serverless Performance

Rust's small, efficient binaries and predictable latency make it ideal for AWS Lambda functions, leading to quicker cold starts and consistent execution.

Enhanced Memory Safety

With Rust, memory-related errors are largely prevented at compile time, significantly reducing a common source of production bugs and improving application stability.

Scalable Asynchronous I/O

Leveraging Tokio, Rust applications can efficiently handle asynchronous I/O, allowing seamless and scalable interaction with AWS services like S3, SQS, and DynamoDB.

Significant Cost Efficiency

Rust's minimal CPU and memory footprints translate directly into lower operational costs for cloud resources, making your AWS deployments more economical.

Robust API Interactions

Strong typing and idiomatic builder patterns in Rust's AWS SDK help prevent misconfigurations and incorrect API calls, leading to more reliable and predictable service integrations.

What Rust Can Improve

Longer Compile Times

Compile times can be longer for large projects, potentially slowing down development cycles.

Steeper Learning Curve

The concepts of ownership and lifetimes present a significant initial hurdle for new developers.

Ecosystem Maturity

While growing rapidly, the ecosystem is still catching up in certain domains compared to more established languages.

Lambda Tooling

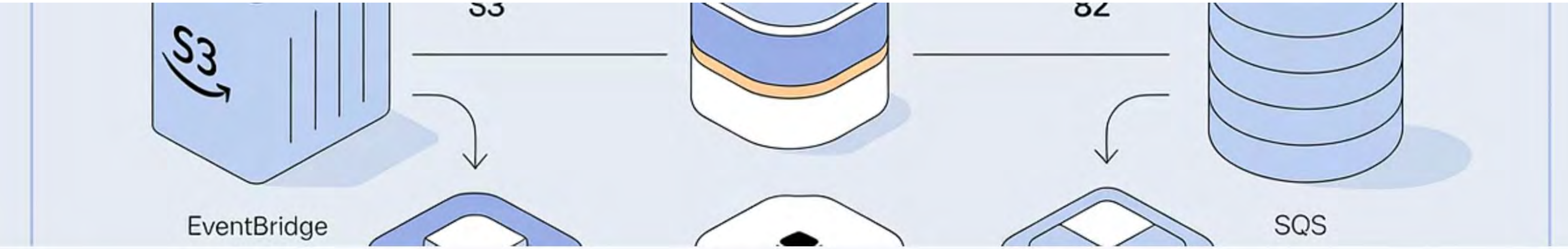
AWS Lambda tooling is improving, but offers fewer "one-click" templates than Node.js or Python.

Limited Metaprogramming

Rust has more limited dynamic metaprogramming capabilities compared to languages like Python or Node.js.

AWS SDK for Rust

1. Modular service crates (S3, DynamoDB, SQS, Lambda)
2. Shared aws-config
3. Async-first with Tokio
4. Built-in retries, pagination, streaming



Flowing Example - File Ingestion System

User Initiates

Users begin the process through system interaction.

S3 Storage

Files are securely uploaded and stored in Amazon S3.

Event Orchestration

S3 upload events are routed by EventBridge.

SQS Notifications

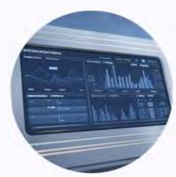
EventBridge forwards messages to an SQS queue for processing.

Lambda Worker

A Lambda function processes the messages from the SQS queue.

DynamoDB Metadata

Processed metadata is stored in DynamoDB for quick access.



Observability

Comprehensive monitoring and logging ensure system health.

Step 1 - Upload Files to S3

- Strongly typed SDK ensures valid requests
- Efficient async uploads

```
// Example Rust S3 upload codeasync fn upload_file( client:
&S3Client, bucket: &str, key: &str, data: Vec) -> Result<()> {
client.put_object() .bucket(bucket) .key(key) .body(data.into())
.send() .await?; Ok(())}
```

Step 2 - Notifications with SQS

S3

EventBridge

SQS

Rust workers consume events safely

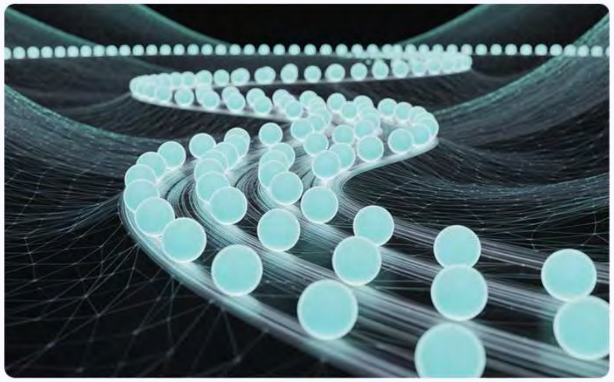
Step 3 - Processing with Lambda

Rapid Cold Starts

<5ms, significantly faster than ~100ms for Node.js. Rust's small binaries enable near-instant function execution.

Minimal Memory Footprint

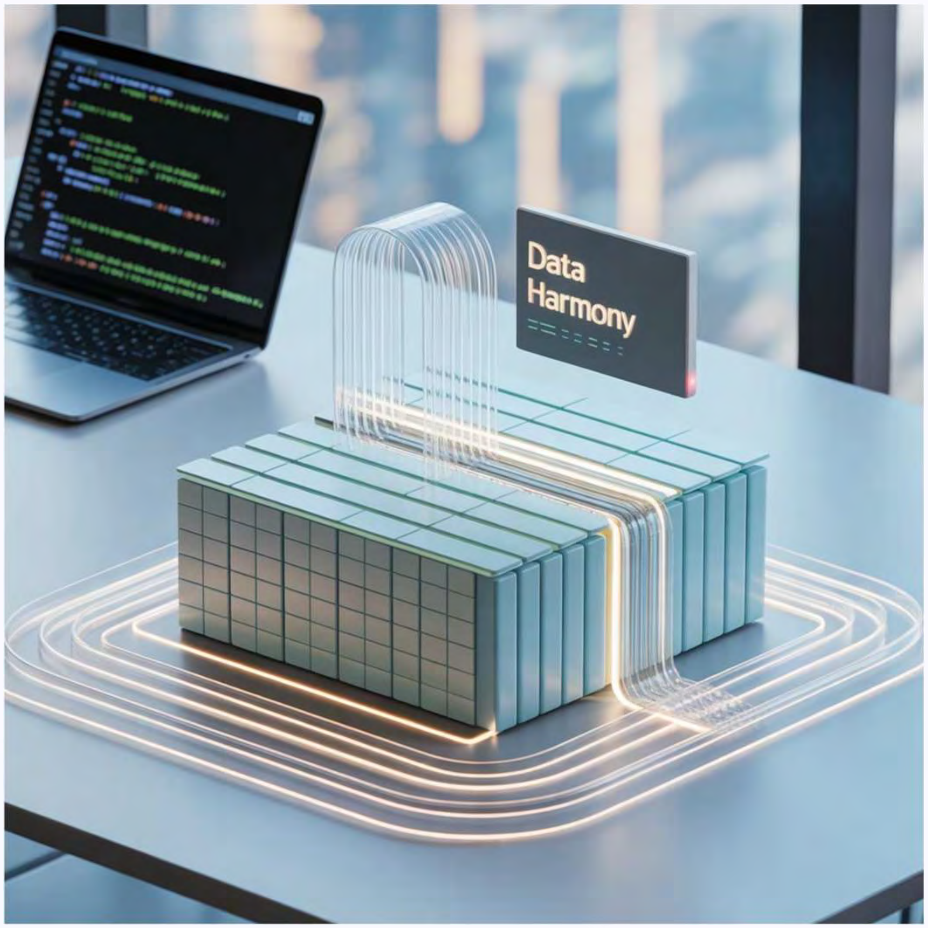
<10MB memory usage, compared to ~50-100MB for JVM. This leads to lower operational costs and better resource utilization.



Predictable Latency

Rust's compile-time memory safety and efficient runtime contribute to stable and predictable performance, even under heavy load.

Step 4 - Store Metadata in DynamoDB



Strongly typed builders for queries

Async operations at scale

```
// Example DynamoDB queryasync fn store_metadata( client:
&DynamoDbClient, table: &str, id: &str, metadata: &Metadata,)
-> Result<()> { client.put_item() .table_name(table)
.item("id", attr_s(id)) .item("data", attr_m(&metadata))
.send() .await?; Ok(())}
```

Step 5 - Observability

tracing for spans/logs

Structured logging with context

OpenTelemetry → CloudWatch

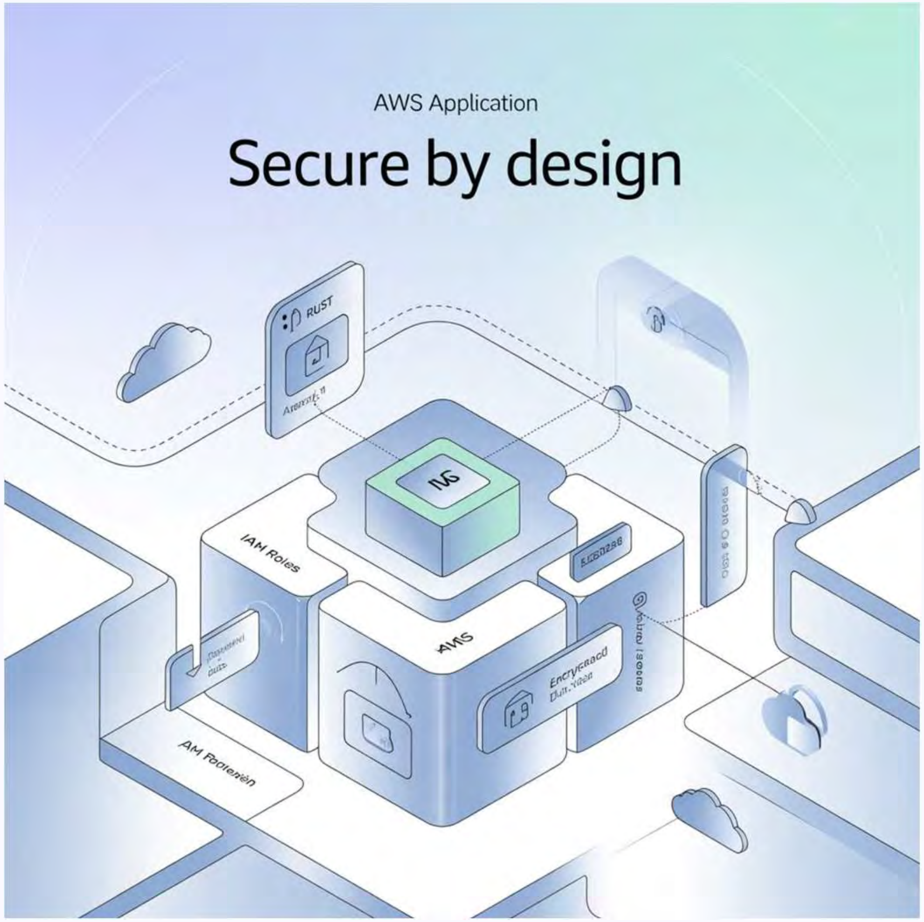
Metrics and distributed tracing

Rich error handling

With anyhow or thiserror

Security & Deployment

- IAM least privilege
- KMS for encryption
- Cross-compile to musl
- Multi-stage container builds



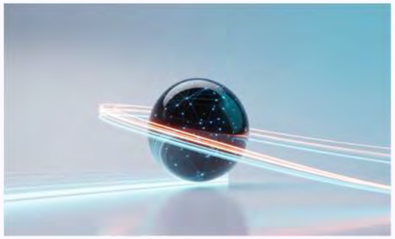
Key Takeaways

1

Compile-time Safety

Leads to significantly fewer runtime bugs.

2



Asynchronous Operations

Enable high I/O efficiency.

3

Modular SDK

Facilitates easy adoption of services.

4

Natural Cloud Patterns

Integrate end-to-end cloud patterns seamlessly.

Rust + AWS = Scalable, Cost-Efficient, Reliable

Thank you!