From Naive to Native

Migrating traditional server-based applications to the cloud and actually reaping the benefits.

MECHANICAL ROCK

Executive Summary

The rise of public cloud computing has been driven by hyper-scale online businesses like Amazon, Netflix and Google. Their platforms were born in the cloud and their success has been in a large part due to the flexibility and scale of cloud computing.

However, many organisations aren't approaching cloud computing with a blank slate. They have existing applications or customers tied to older software architectures.

These organisations tend to have lacklustre or lukewarm experiences when moving to the cloud - failing to see the kind of benefits and cost savings they expect. In a 2019 paper from Unisys about 39% of businesses in Australia rated their cloud migration as "below expectations".

While re-architecting for the cloud requires more investment, the value returned over time far exceeds the cost. Further, the cost can be spread over an extended period if you methodically tackle the problem and adopt new technologies in a step-by-step fashion.

The Path to Value

This e-book maps out a better path for organisations who develop or use custom software to reap higher value from cloud computing.

Traditional Server Based Applications

Over the past 50 years the composition and nature of technology used in software applications has changed radically. Each stage of development from mainframe, to client server to service oriented architecture to the modern web stack has meant different tradeoffs and compromises.

While all of these may run on modern cloud infrastructure, modern cloud services have been designed with the highly parallel, asynchronous world of web applications in mind. Using a suboptimal architecture results in suboptimal outcomes in terms of cost, performance, reliability and the ability to innovate quickly.

To reap the benefits of public cloud computing, owners of traditional server based software should look to modernise their architectures in a structured way.

Common Architectural Patterns

Client-server	Also n-tier or 3-tier consisting of data, logic and presentation tiers.	Traditionally featured synchronous comms and monolithic applications that limit or constrain improvements due to cross-tier dependencies.
MVC	Model-View-Controller - a specific implementation of the 3 tier model.	Associated with GUI interfaces, separated presentation from 'domain model' but has become complicated and has limited utility in other settings.
SOA	Service Oriented Architecture	Separates 'services' with a common message protocol or bus to reduce coupling but often ends up hiding the coupling in the message bus.
Web server	REST GraphQL	Developed to suit the distributed World Wide Web, with loose coupling and lightweight messaging between services, REST is the original web server standard while GraphQL is a newer standard, optimised for certain use cases.

Common Technology Stacks

Lamp Stack	Linux Apache Perl, PHP or Python MySQL	Common web stack - simple, flexible and easy to use, particularly with respect to the programming language used (Perl, Python, PHP). Often still requires JavaScript to implement web front end, which adds complexity.
Ruby on Rails	Ruby / Rails SQL database (Apache Web Server)	An MVC implementation which uses Ruby (a python like language) and the Rails framework to implement "CRUD" operations against a database.
WAMP WIMP WINS	Windows Apache .NET SQL server	A version of the LAMP stack, replacing Linux with Windows and the P-languages with Microsoft's .net framework and a language like C#, VB or J#.
MEAN MERN MEVN	MongoDB Express.js Angular/React/Vue Node.js	JavaScript based stack with a MongoDB noSQL database replacing the SQL component of a LAMP/WAMP stack. Angular, Node, React & Vue are all JavaScript frameworks.

Modern Application Patterns

With the rise of ubiquitous compute resources in the form of public cloud services, a number of new patterns for software architecture have developed. Exemplified in the hyperscale web apps or Software-as-a-Service platforms they are surprisingly simple and flexible.

The philosophy which underpins modern software architectures is the 12-Factor App methodology. This highlights the key principles for building distributed software services that are scalable, resilient and efficient.

The implementation of these principles can be achieved with the following architectural patterns:

Microservices	Database per Service Saga Event Sourced Service Mesh	The rise of internet scale services (like Netflix or Amazon) gave rise to architectural patterns where applications are distributed over many small services, that scale horizontally and can be developed independently by different teams. This leads to some challenges but typically provides faster development, better resilience and agility.
Serverless/FaaS	AWS Lambda Azure Functions Google Cloud Functions Open FaaS	One way to implement microservices is using a cloud based “Function as a Service” model. This uses ephemeral compute resources to run functions on-demand. It is extremely cost efficient with minimal overhead and incredible flexibility.
Containers	Docker AWS ECS on Fargate Kubernetes AWS EKS Google GKS Apache Mesos	Building on the success of “virtual machines” to provision hardware, containers are an abstraction that includes the run-time an application requires. This means that the same container which runs on a developer’s laptop, can run on a cloud server, or a cloud orchestration platform like Kubernetes. This makes deployments easy, reliable and scalable.
Data stores	MongoDB DynamoDB AWS S3 Neptune MariaDB Redshift BigQuery Snowflake	Traditionally data has been stored in SQL databases for the optimal performance of specific types of access. But as more types of data became common and compute prices have fallen, different types of data store have been developed optimised for different use cases. Unstructured data, network databases, object stores and data warehouses have come to dominate in places other than OLTP where SQL still reigns supreme.
Front-end clients & API frameworks	Progressive Web App Javascript frameworks Web Assembly GraphQL React Native CDNs	Content Delivery Networks (CDNs) offer global scale and availability for web applications at the fraction of the cost of traditional server hosting. Modern JavaScript frameworks, like React, are designed with this in mind to support rich interactive user experiences. WebAssembly (Wasm) is a new standard offering a modern web architecture using alternate languages, such as C#. Progressive Web Applications (PWAs) offer similar functionality to native mobile applications without the overhead of separate code bases for web, desktop and mobile.

See also: *Microservice Architecture patterns*

The Application Modernisation Journey

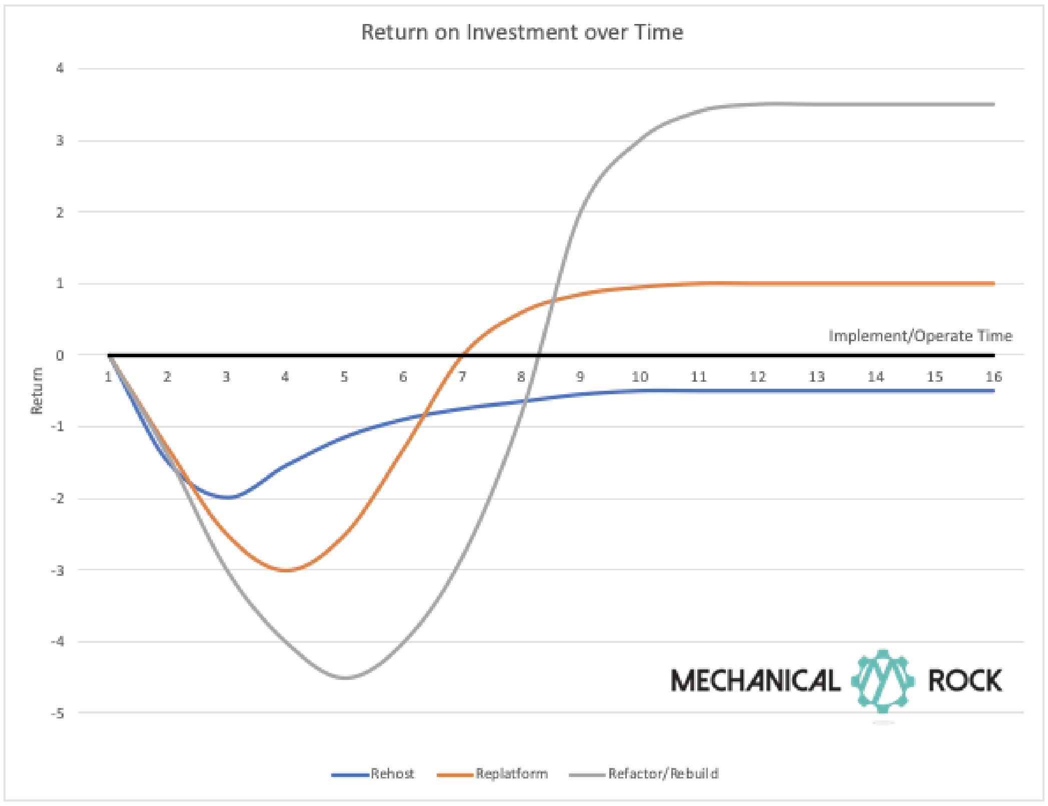
In general there are six choices when migrating applications to the cloud.

For custom built software, only three of them usefully apply:

1. **Rehost** in the cloud using the same processes and technology as on-prem;
2. **Replatform** by automating some of the underlying services or using managed services;
3. **Rearchitect** the application to take advantage of higher levels of cloud maturity.

Ultimately the choice is based on the return on investment you wish to realise.

While re-architecting for the cloud requires more investment, the value returned over time far exceeds the cost. Further, the cost can be spread over an extended period as you break apart the problem and adopt new technologies in a step-by-step fashion.



The next section of this e-book lays out a methodical approach to modernising your application or software in the cloud.

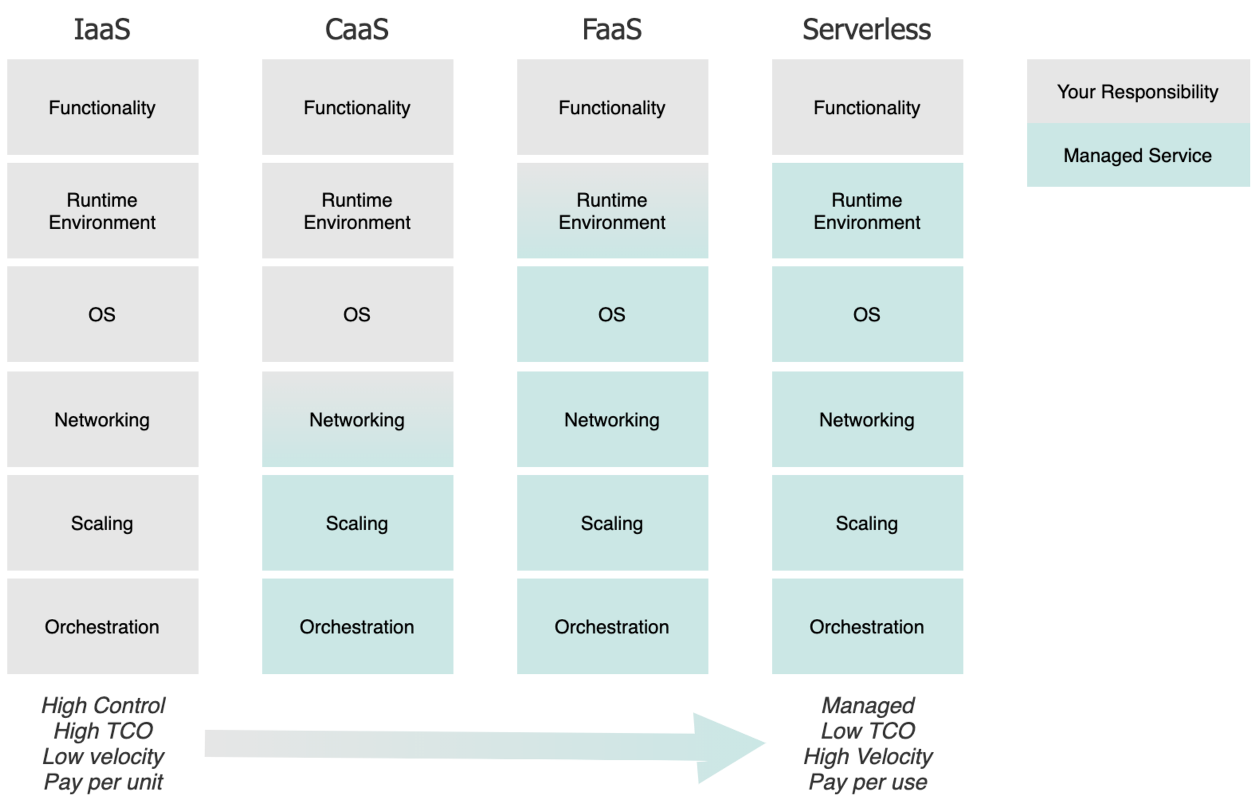
Technology Adoption Roadmap

It is important to learn how to manage your application effectively in the cloud before attempting to re-architect and optimise. By gathering data as you go you will be able to make the most effective decisions about how and where to optimise your architecture.

A serverless, cloud native, system is recommended for the long-term target architecture in order to minimise the total cost of ownership (TCO) and maximise reliability and development velocity.

The first step in the journey is to enable repeatable and reliable infrastructure through infrastructure-as-code, configured and deployed with an automated pipeline.

Then you should migrate to hybrid components, like an orchestrated container solution, to support a “Function-as-a-Service” deployment model. These can be implemented at a lower cost, can be continuously optimised and will enable a longer term re-architecture to a serveless model.



Ultimately, the higher levels of cloud maturity offer compelling benefits:

- Developer effort can be redirected from the overhead of managing ‘plumbing’ to adding business value or delighting customers.
- Cost efficiency is achieved by the metered usage of easily scalable resources.
- Decomposition of services means they can be updated quickly and optimised individually. They can also be distributed geographically to increase reliability or reduce latency.

Cloud Maturity Ladder

Many organisations have had lacklustre experiences with cloud - failing to see the benefits and cost savings they expect. In a 2019 paper from Unisys, 39% of businesses in Australia rated their cloud migration as "below expectations".

This is mainly because many treat public cloud services as 'just another data centre' - which fundamentally misses the point. Public cloud services evolved because existing data centres required a level of overhead and toil that companies like Amazon or Google could not sustain. Modern public cloud services all feature a layered service model designed to remove as much of the 'undifferentiated heavy lifting' as possible.

A failure to appreciate and leverage higher levels of cloud services will deliver disappointing results. Accessing these higher levels means gradually changing processes, capability and tooling in a methodical way over time.

	Cloud Naive	Cloud Optimised	Cloud Native
Tooling	Common tooling across cloud & on-prem.	New services built in the cloud with tooling selected to match, with an eye on longer term evolution.	Tooling optimised for target environments based on factors such as cost, performance and scale.
Deployment	Infrastructure provisioning separated from application deployment and based on service requests.	Self service provisioning. Optimised deployment to reduce lead times. Deployment in smaller batches to 'lower the water level and expose the rocks'.	Deployment straight to production with zero down time utilising feature flags to separate release from deploy; immutable infrastructure, all deployed 'as-code'; canary releases; automated rollback.
Components	Compatible managed services used as a drop-in replacement for existing components.	System redesigned to support chosen tenancy model. Strangler pattern* used to migrate remaining services to managed or cloud native alternatives.	System redesigned to event driven architecture. Hybrid FaaS architecture, with Lambda for deployments. Serverless PAYG cloud services to minimise costs.
Service Optimisation	Optimisation based on simple t-shirt sizing, results in unused capacity over time.	Redesigned for horizontal scalability. Supports scale to zero where appropriate.	Only cloud native services for cloud components - no servers. Service is self optimising and self healing.
Cost Model	Resource based	Usage based (wherever possible)	Economics measured on a service-by-service basis. Refactoring for cost savings.
Cost Optimisation	Basic utilisation and cost tracking. Focus is on fixed capacity planning, using things like reserved instances.	Resource allocation is optimised based on usage. Savings maximised on static resource usage.	FinOps - automated cost monitoring and real time optimisation of costs. Experiments in refactoring to minimise TCO or improve reliability and performance.

**Strangler Pattern - an architectural pattern that uses a facade or message tap to duplicate traffic to a service as a means to rewrite that service in place. Named after the Australian strangler fig.*

Tenancy Model

The tenancy model is a key decision and determines the approach for managing the overall cloud footprint and in determining the architecture of the rest of the application. Tenancy refers to the number of customers that share resources at any given level of the application stack.

As with all design choices, there are tradeoffs to be made:

Single Tenant	Single compute instance (eg. server, container, VM) per customer. Single data store or database per customer All services and infrastructure are configured on a per customer basis, with complete segregation from any other customer.
Hybrid Tenant	Shared compute orchestration platform (e.g. Kubernetes cluster) -or- Shared data service (e.g managed database) but segregated data.
Multi Tenant	Shared compute & data instances (eg. microservices cluster, managed database) All services and infrastructure are shared between customers, with security policies and authorisation flows defining access control.

At a high level, the differences between multi and single tenant are:

	Multi Tenancy	Single Tenancy
Scalability	✅ Potentially unlimited scale	❌ Scale capped by cost and effort
Cost	✅ Economies of scale	❌ Cost scales linearly with number of clients, effort scales exponentially.
Utilisation	✅ Much better resources utilisation	❌ Duplicate resources per client.
Speed	✅ Faster deployment of changes	❌ Complex, burdensome release cycle
Segregation	= Segregation handled within the application and data layers.	= Segregation based on infrastructure configuration.
Customisation	❌ Customisation possible through config, but limited by shared model	?? Easily implemented, but increasingly costly as customers scale.
Blast Radius	❌ Wide blast radius, requires discipline	✅ Small blast radius,
Complexity	❌ More complex application architecture	✅ Simplified application architecture

A hybrid model is a choice based on how far along the scale is appropriate for your requirements. For example, resource utilisation may not be a concern, so deploying distinct infrastructure per customer may be feasible, but it will have significant cost implications.

Cloud Governance Strategy

Part of the benefit of cloud services is through distributed self-service models of delivery, (i.e. teams can provision the resources they need through self service-automation). This avoids delays and bottlenecks and enables speed, experimentation and innovation.

However if you deploy self-service options but retain a centralised governance structure then it will become a bottleneck which will throttle your productivity. The answer is to develop a rules based approach which delegates authority appropriately in line with your business objectives.

Your cloud governance model should be layered, with appropriate controls at each level:

Operational design	Cloud Operating Model	Monitoring, logging and observability	Capacity planning and resource optimisation	Resilience, disaster recovery and business continuity
Workload design	Account vending (inc guardrails)	Infrastructure deployment patterns	Application deployment patterns	Compute and Storage options
Cloud Foundations	Organisational Guardrails	Network design & cloud connectivity	Account topology	Identity Management & Access Control

Cloud Foundations

Organisational Guardrails	Cost and billing, organisational policies and universal security controls.
Network design & connectivity	Network design including connectivity to on-premise (if required).
Account topology	Grouping using organisational units and accounts as workload boundaries to limit blast radius (see below).
Identity Management and Access control	Federated identity management, zones of control and access role definitions provide layered access controls.

Workload Design

Account vending (inc guardrails)	Incorporating guardrails into the (automated) account vending process ensures workload controls are deployed automatically at creation.
Infrastructure deployment	Deploying infrastructure-as-code via automated pipelines (possibly self service) ensures consistency and control and minimises 'drift'.
Application deployment	The deployment of applications and services should be automated to the highest extent possible using cloud based services.
Compute & storage options	Cloud services offer numerous choices for both compute and storage, with costs and benefits. Principles governing their use should be established to ensure business goals are met.

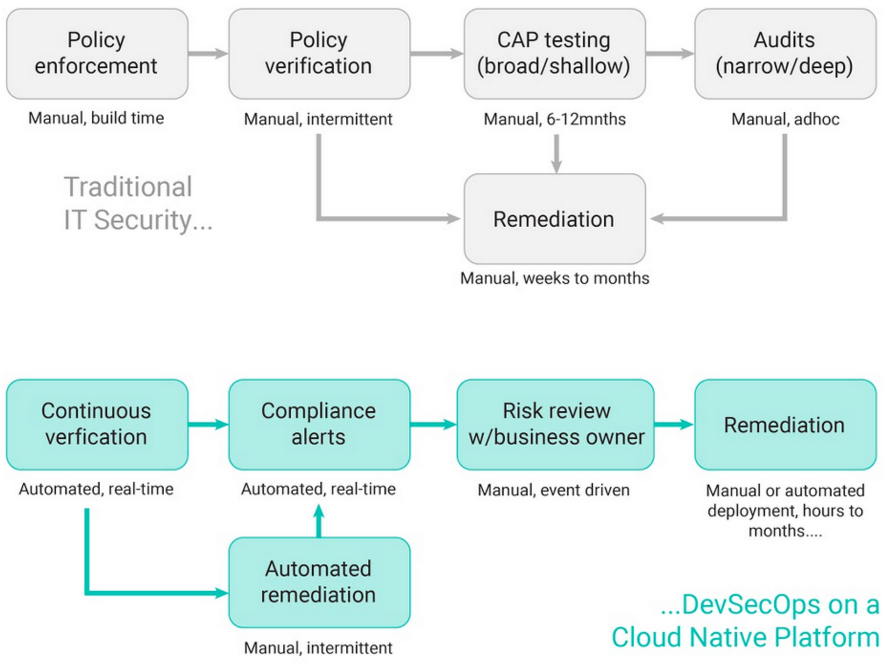
Operational Design

Cloud operating model	Moving to the cloud represents a unique opportunity to redesign organisational responsibilities to provide a more flexible, but secure operating environment.
Monitoring, logging & observability	Visibility and transparency of activities and operations is key to ensuring control is achieved and exceptions are minimised.
Capacity planning and resource optimisation	The cloud brings new challenges and opportunities in capacity and demand management
Resilience, disaster recovery & business continuity	Flexible, on-demand resources mean higher levels of resilience can be achieved at lower cost, but only by adopting cloud native principles.

Organisational Guardrails

Guardrails are categorised into:

1. Preventative Controls: prevent security events from occurring
2. Detective controls: detect security events when they occur
3. Remediation controls: react to security events to minimise their effect in a timely manner



Defense-in-Depth

Cloud foundation controls and guardrails are applied at a number of layers:

- Baseline controls: These are guardrails that you wish to apply to all accounts, plus any accounts you create in the future
- Core patterns: These are patterns you wish to apply to the organisation, but what you apply and where may vary. Examples include:
  - Security cross-account roles
  - Central log auditing
  - Shared networking
- Workload patterns: These are patterns you wish to apply to a particular type of workload.

Typical organisational guardrails that should be enabled include:

- Geographic controls that limit where workloads can be spun up (preventative/baseline)
- Prevent public access to object storage (preventative/baseline)
- Whitelist allowed services in workload accounts (preventative/core pattern)
- Enable detective controls like AWS Guard Duty across all accounts (detective/baseline)
- Enable budgets and thresholds (detective/core pattern)
- Automatically enable logging for new accounts & services (detective/core pattern)

Account Topology

There are multiple ways to set up AWS accounts within an organisation and costs and benefits to each approach. Note that an account structure is related to the tenancy model you select - the account model determines access and blast radius at the infrastructure level, while the tenancy model determines shared dependencies at the application level.

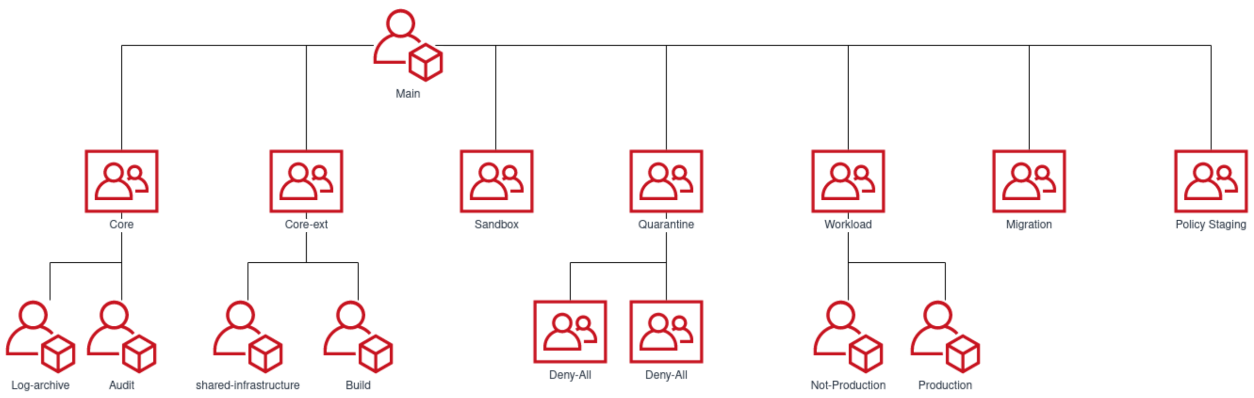
For example, a multi-tenant application could be spread across multiple accounts, with each account being geographically dependent (i.e. all the customers in a specific region).

In general follow these principles:

- Group workloads based on business purpose and ownership
- Apply distinct security controls by environment and risk profile
- Constrain access to sensitive data
- Limit the scope of impact from adverse events
- Promote DevOps operating models where all the resources required to build and operate a workload are located within an account group which is available to a single team

Recommended Account Structure

Using AWS Organisational Units, you can further refine the structure of your account model:



Cloud Governance team			Development & Operations teams	
Core	Core-ext	Quarantine	Workload	Sandbox
Security logging and audit	Cloud ops and workload management via shared services.	Holding area for suspended accounts, prior to closure. Break-glass lock-down controls in case of a breach.	Prod/staging pairs of accounts per team to limit blast radius but enable delivery of value.	For experiments to encourage innovation and cloud capability maturity.

DevOps Ways of Working

DevOps provides developers agility so they can deliver value to customers through the rapid delivery of software.

In principle this means *an account per workload or application*.

This has a number of benefits:

1. Using a separate account *per team* or *per product* limits the blast radius of issues with that product to that product. For example, a change in the deployment pipeline or access controls for a particular product could bleed over into other products within the same account; a mistake in a deployment script could deny services to all applications which share the deployed services. Account separation prevents this.
2. Further a separate account per workload makes that workload *portable*. Because accounts are effectively self-contained, if a workload ever transitions between teams, or teams are amalgamated, then you only need to handover the keys to that account and they have everything they need to develop and operate that workload.
3. Finally tooling can be specific to the workload. This allows development teams the flexibility to *choose their own stack, tools and methods*, further increasing their agility. Centralised platform or tooling teams often become bottlenecks in their own right.

From a business perspective the separation of accounts allows costs to be tracked on a per-project/product basis. This enables investment decisions to be made with a much clearer line of sight to the operational costs associated with each system, and to consequently measure against benefits. It also allows direct comparisons between products and enables experiments to improve throughput within a particular product chain.

When managing large numbers of VMs, running COTs software, it is recommended to host these within an account per environment: such applications have minimal inter-dependencies and are usually managed by a centralised IT team - the overhead of managing multiple accounts outweighs the benefits in this instance. A "shared services" account is useful for these purposes.

Sandbox Accounts

Sandbox accounts provide a cloud playground to promote experimentation and capability development. Sandboxes can be assigned to individuals; pooled or shared within teams.

‘Innovation budgets’ using budget alerting is a simple way to manage costs without onerous approval processes: allocate an approved spend for experimentation. If the limit is exceeded, further approval or cleanup of unused resources is required.

Sandboxes should be fully isolated with no access to sensitive data - they are for experimentation purposes only.

Cautionary Tales from the Field

Given our experience with clients we have seen a number of examples where poor account structures have severely limited their ability to utilise cloud services.

1. A large enterprise client with a single AWS account and a dozen teams would hit service limits on a monthly basis (e.g. network service limits) and while these are soft limits that can be altered, each time teams lost multiple days diagnosing and resolving issues through support channels.
2. A smaller client ran all their workloads in the same account. Consequently there were multiple instances where developers deleted resources by accident. The most significant was where a developer accidentally deleted a production database table. Although it could be easily restored, it represented unnecessary risk, anxiety and effort.
3. We have experienced many examples of clients with multiple tenants or workloads in the same account where one team has deleted or reconfigured a shared resource (like a VPC endpoint) or even mistaken another team's services for their own and broken multiple systems, including production. Accounts offer a clear boundary for delineation that aligns well with team and product boundaries.
4. A smaller client migrated on premise VMs to the cloud. The COTS nature of many products and the fact that applications were managed by a centralised support team led to the workloads being co-located in a single "service" account with guardrails. In this instance, separating each VM into a separate workload account would have added unnecessary management overheads.

User Access and Identity Management

Manage access and identity centrally using a federated model based on existing organisational directory services. Replicate on premise directories to the cloud as a preparatory step, e.g using Azure AD, to support a modern standards based approach and phased migration.

SAML and OpenID Connect (OIDC) reduce vendor lock-in and support the use of Single Sign On (SSO) services.

Design Role Based Access Controls (RBAC) or Attribute Based Access Control (ABAC), to align user permissions with your organisational policies, mapping to the native Identity Management controls of your cloud provider. Manage system-to-system access control using cloud native controls to eliminate the need for secrets management and enforce a least-privilege model.

In general, we recommend:

- Simple, light touch controls to reduce management overheads
- Empowering users over controlling their actions
- Auditing and engagement over prevention and approvals
- Delegate permissions to minimise handoffs, and maximise flow:
- Separating responsibilities through peer review (PR) rather than job functions
- Avoid manual error by automating flows wherever possible

An effective identity management policy is important to control the organisational risk of unauthorised, or malicious, access. However, a one size fits all policy is not appropriate as complex identity management policies are overly restrictive and costly to maintain. As an organisation grows and the number of projects and value increases, the risk associated with a permissive structure grows.

Organisation Risk/Challenge	Mitigation	Zone of Control
A compromised user account has access to multiple accounts	Segregate account access based on team role.	Team membership, mapped to account access.
Users have permission not required for their function	Maintain user groups/attributes based on job function.	Job function / account permissions
User permissions are abused	Maintain an audit log of activity for post-mortem analysis	Audit / Monitoring
Multi account strategy makes identity management difficult	Automate self-service provisioning and group access. Centralise identity management and apply consistent onboarding/offboarding process.	Self service provisioning Personnel management processes
Increased complexity with multiple accounts and complex organisational hierarchy	Avoid tightly coupling the account strategy to the existing business unit structure. Avoid complex (deeply nested) organisation hierarchies. Align accounts with services provided and the teams responsible for them.	Team/Organisation membership
Privilege escalation	Control access to creation of users and roles, and associated permissions.	Permission Boundaries / Service Control Policies

Continuous Flow of Value

While the foundations of networking, accounts and infrastructure are essential they should be regarded as 'plumbing' which supports the true value of your digital enterprise – the applications that business users and customers require.

To that end, not only should the foundations be designed to operate with as little overhead as possible, they should be designed to support a flow of value in your organisation. They should be designed to allow the rapid and easy deployment of changes to infrastructure & applications.

The State of DevOps Report from Devops Research Associates (DORA) has directly linked Software Delivery and Operational (SDO) performance with organisational performance metrics like profitability and speed to market.

Further the report clearly identifies four key metrics which are proven to drive Software Delivery and Operational performance :

Measure	Description	Indicator of:	Elite Performance:
Lead Time	How long does it take a change to go from code to running in production?	Speed Efficiency	< 1 day
Deployment Frequency	How often does your organization deploy code to users?	Efficiency Risk	On-demand (multiple per day)
Change Failure Rate	What percentage of changes to production result in degraded service?	Stability Reliability	0 - 15%
Time to Restore	How long does it take to restore service when a defect impacts users?	Reliability Availability	< 1 hour

In order to maximise the long-term value you derive from your transition to a cloud architecture, you should:

1. Collect historical data for these metrics based on your pre-cloud baseline
2. Design your software delivery architecture with an eye to a step change improvement
3. Build automated collection and measurement for each metric into your cloud reporting
4. Set short and medium term improvement goals for each metric
5. Implement small scale experiments as 'proof of value' evaluations
6. Roll out successful experiments horizontally throughout your organisation
7. Go to step 4 and repeat.

Infrastructure Deployment Patterns

The manual process of deploying, configuring and operating temperamental hardware has been largely replaced by commodity scale 'infrastructure provisioned 'as-code' (IaaC).

The automation of infrastructure provisioning offers many benefits:

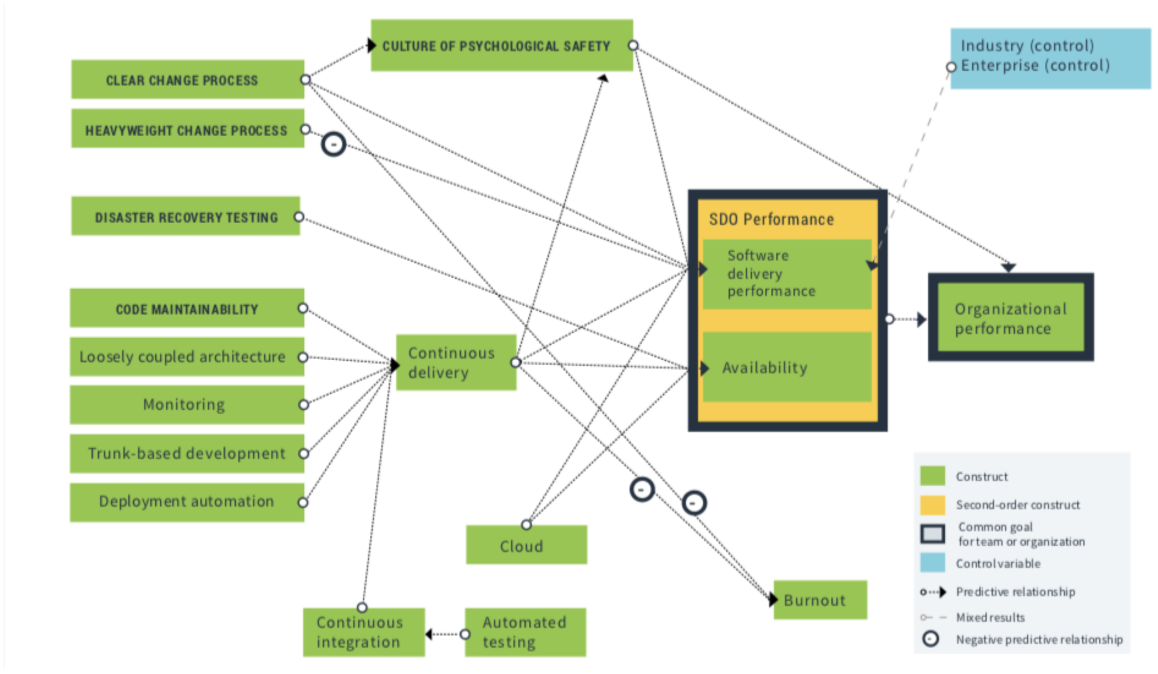
- Enables rapid provisioning and tear-down so environments can be 'on-demand'
- Ensures consistency between deployments, removes manual error
- Improves consistency between different environments and reduces 'configuration drift'
- Reduces risk by eliminating over-privileged manual access

The use of automation leads to 'immutable infrastructure' - where infrastructure is always recreated from a defined set of rules (code) when it is required. Changes are always applied to the defined rules and never to the deployed infrastructure. The opposite of immutable infrastructure are 'snowflake servers', where each server is complex and unique.

Use a centralised Version Control System	Once you have adopted 'infrastructure-as-code' you can control changes to infrastructure through the same mechanisms used for software - a version control system. This allows engineers to collaborate on changes and adopt rigorous software engineering practices like code reviews and automated testing. Ideally the infrastructure code should be controlled alongside application code to ensure the two are deployed in sync and to avoid unintentional configuration drift.
Account factory	The basic workload boundary in the cloud is an 'account'. Organisations should have many accounts, separated on a logical basis and provisioned automatically, including the appropriate guardrails. By customising guardrails and infrastructure based on the target workload you can create an account 'factory' which allows teams to provision their own accounts.
Product catalogues	An analog to an account factory is a 'product catalogue' where various workload infrastructure patterns (e.g. a serverless web application) can be stored, along with appropriate security controls. Teams that have created accounts can then self-service infrastructure provisioning from an approved set of patterns.
Modular Infrastructure Code	Instead of defining your workload infrastructure in a single bit script, design individual modular components that can be deployed in the correct order to build different services. Abstract configuration details into variables or parameters to ensure flexibility.
Apply Automated Testing Patterns	Continuous integration and delivery (CI/CD) in software relies on automated testing to verify correctness. The same principle can be applied to infrastructure. Use unit and integration tests to test the behaviour of deployable infrastructure to ensure it conforms to the desired outcome (this is known as Behaviour Driven Infrastructure or BDI, see Behaviour Driven Development or BDD for comparison).
Zero trust networking	Zero trust networking eliminates trust from an organization's network architecture and replaces it with a continuous assurance of trust. A zero-trust model considers all resources to be external and continuously verifies trust relationships before granting only the required access. This provides continuous protection from both external and 'internal' resources (link).

Software Development Patterns

The following diagram is taken from the DORA State of DevOps Report. It depicts the strong causal relationship between certain software development practices and Software Development and Operational (SDO) performance.



These practices are fundamental to supporting the continual flow of value in a digital enterprise.

Continuous Integration	Code changes that are not in production are an overhead that requires time and resources to manage. They also represent an unknown risk until they are deployed and tested.To increase reliability and reduce the lead time to production it is recommended that code changes are merged into working software at least daily, if not *several times per day* (link).
Continuous Deployment	The logical extension is to continuously deploy software changes into production. Note, this does not necessarily mean that every change is immediately visible to users since the release of functionality can be governed by configuration (feature flags). It will mean however that you shorten your feedback loops, minimise rework and increase the reliability of your deployment chain as you will repeat it frequently and deal with issues as they arise (link).
Blue/Green Deployment	The use of continuous deployment and distributed compute models (like microservices) enables changes to be made in real time with no outages. By adopting a blue/green deployment model where compute nodes are gradually refreshed with code changes you can seamlessly migrate your used base from one version of software to the next (link).
Canary Deployment	A variation of the blue-green deployment model is the canary model where a change is released to a subset of users. The change can then be evaluated in production and, if successful, rolled out across the user base. This enables experimentation and innovation.
Trunk Based Development	A fundamental development practice which enables continuous integration and deployment is “trunk based development”. In TBD developers merge their changes into the ‘mainline’ or trunk code several times per day. This encourages the integration of smaller changes, more frequently, which in-turn enables continuous integration and deployment.
Evolutionary Database Design	A limiting factor on deployment can often be database coupling which limits code changes as the database is shared across versions of the software. To make this work design must be treated as an on-going process that is interleaved with development, testing and delivery rather than a separate step - this is known as evolutionary design. (link)

Design for Continuous Operations

In order to provide an exceptional user experience, understanding how your systems behave and proactively addressing issues before they escalate is crucial. Antifragile systems remove the need for fire-fighting, enabling focus on new features and delivering value. Improved feedback loops allow you to understand usage patterns and to ensure you focus on building valuable services.

Monitoring and Observability

Typically on-premise hardware and software assets are tracked via cumbersome methods like a manual "CMDB" and logging and monitoring are an afterthought. With cloud computing much of this comes for free and offers much greater transparency but careful thought must be put into its design, configuration and deployment (generally automated).

Appropriate tools are required for observability, to provide Logging, Monitoring and Tracing to:

- Measure the Mean Time to Recovery (MTTR) and Mean Time Between Failure (MTBF)
- Reduce the time for problem identification (MTTI), and problem resolution
- Enable fault diagnosis, troubleshooting and continuous improvement
- Satisfy security requirements for logging and auditing

The following targets represent 'best-of-breed':

Problem Identification	When a failure occurs, it should be possible to identify the cause of the failure within 10 minutes. When a failure occurs, it should be possible to perform analytics in order to identify the root cause within 4 hours.
Alerting	It should be possible to configure automated alerts to notify failures. It should be possible to perform automated responses based on the alerts.
Access Control	Application logs can contain sensitive information. It should be possible to configure access control for managing access to logged information. Access log themselves should be secured to only authorised individual
Delegated Responsibility	Application teams need to be able to self manage and control access to logs on an operational-needs basis.
Service Level Objectives and Error Budgets	Applications can track and report SLOs to help determine where to prioritise engineering work..SLOs include: MTTR; MTBF; latency, saturation, traffic, and errors. Error budgets enable a calculation of acceptable failure and assist planning. If the error budget is, or is likely to be, exceeded then remediation effort is required..
Application SLIs	Service Level Indicators (SLIs) provide a measure of service based on user-centric metrics: Successful request ratio, cache miss ratio, ratio of page loads < 100ms, etc.

Rules of Thumb

- Always ensure logs contain sufficient log event data to address the specific requirement
- Access logs should support both success and failure of specified security events
- Ensure log entries that include un-trusted data will not execute as code
- Do not store sensitive information, including session identifiers or passwords
- Log all apparent tampering events, including unexpected changes to state data
- Log attempts to connect with invalid or expired session tokens
- Log all administrative functions, including changes to the security configuration settings
- Use a cryptographic hash function to validate log entry integrity

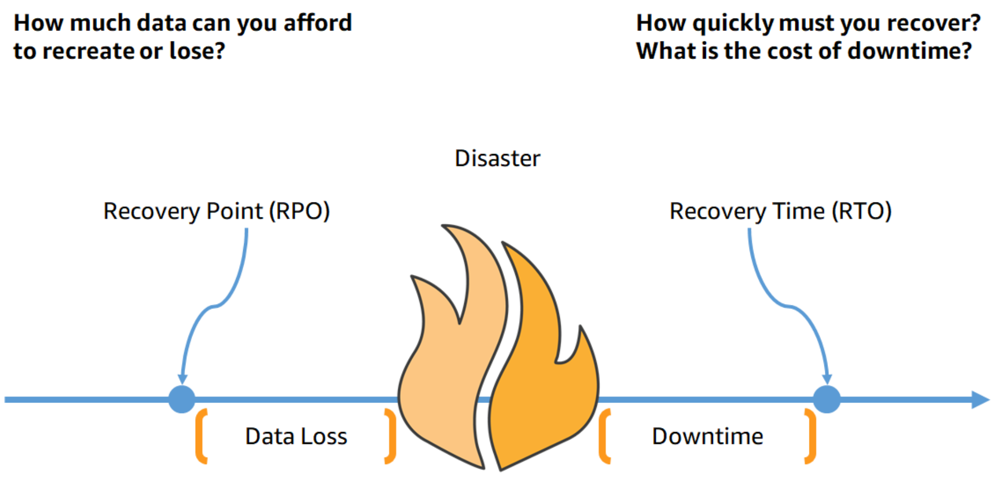
Disaster Recovery

Disaster recovery is the process of preparing for and recovering from a disaster. An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location is considered a disaster. The aim is to mitigate risks and meet the Recovery Time Objective (RTO) and Recovery Point Objective (RPO) for a particular workload.

For more detailed insight into disaster recovery in an AWS context read: *Disaster Recovery of Workloads on AWS: Recovery in the Cloud* Whitepaper..

Recovery Objectives (RTO and RPO)

When creating a Disaster Recovery (DR) strategy, organisations most commonly plan for the Recovery Time Objective (RTO) and Recovery Point Objective (RPO).


How much data can you afford to recreate or lose?
How quickly must you recover? What is the cost of downtime?
Disaster
Recovery Point (RPO)
Recovery Time (RTO)
Data Loss
Downtime

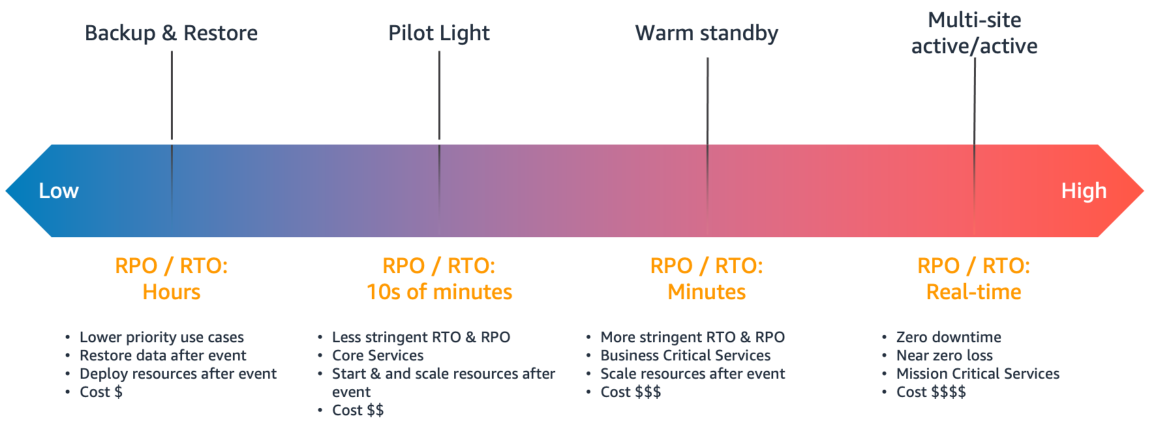
Recovery Point Objective (RPO) is the maximum acceptable amount of time since the last data recovery point. This objective determines what is considered an acceptable loss of data between the last recovery point and the interruption of service and is defined by the organization.

Recovery Time Objective (RTO) is the maximum acceptable delay between the interruption of service and restoration of service. This objective determines what is considered an acceptable time window when service is unavailable and is defined by the organization.

Disaster Recovery Patterns in AWS

Disaster recovery strategies available to you within AWS can be broadly categorised into four approaches. For traditional server based applications they represent a spectrum of cost and complexity, but the higher levels are available to modern cloud native solutions at no extra cost.

It is also critical to regularly test your disaster recovery strategy so that you have confidence in invoking it, should it become necessary.



- Lower priority use cases
- Restore data after event
- Deploy resources after event
- Cost $

- Less stringent RTO & RPO
- Core Services
- Start & and scale resources after event
- Cost $$

- More stringent RTO & RPO
- Business Critical Services
- Scale resources after event
- Cost $$$

- Zero downtime
- Near zero loss
- Mission Critical Services
- Cost $$$$

Backup and Restore

Backup and restore is an effective approach for mitigating against data loss or corruption when you have strong IaC practices. Consistent deployment pipelines and immutable infrastructure enable environment recreation in hours.

In the event of a full region failure, services would be unavailable for the duration of the outage. Your workload data will require a backup strategy that runs periodically or continuously and aligns to meet your RPO.

Pilot Light & Warm Standby

These two strategies focus on multi-region failover scenarios whereby resources are standing by in another region to be scaled up in the event of a failure. Strong IaC and DevOps practices will enable environment recreation in the space of hours (pilot light) to minutes (warm standby).

Multi-site active/active

You can run your workload simultaneously in multiple regions as part of a *multi-site active/active* or *hot standby active/passive* strategy. Multi-site active/active serves traffic from all regions to which it is deployed, whereas hot standby serves traffic only from a single region, and the other region(s) are only used for disaster recovery.

While multi-site active/active may seem complex and costly, if you architect your system to use distributed scalable services (containers, serverless etc) then it can run in multiple regions simultaneously and will provide the highest levels of resilience and continuity.