

Power Platform – Well Architected Framework (WAF)

# Contents

[Contents 2](#_Toc179192184)

[1 Introduction 4](#_Toc179192185)

[1.1 Purpose 4](#_Toc179192186)

[1.2 Scope 4](#_Toc179192187)

[1.3 Areas of Scope 4](#_Toc179192188)

[2 Reliability 5](#_Toc179192189)

[2.1 Pillars 5](#_Toc179192190)

[2.2 Design for Business Requirements 5](#_Toc179192191)

[2.3 Design for Resilience 6](#_Toc179192192)

[2.4 Design for Recovery 7](#_Toc179192193)

[2.5 Design for Operations 8](#_Toc179192194)

[2.6 Keep it Simple 9](#_Toc179192195)

[2.7 Checklist 10](#_Toc179192196)

[2.8 Trade-offs 11](#_Toc179192197)

[2.9 Review Design Principles 13](#_Toc179192198)

[2.10 Set and Measure Reliability Targets 13](#_Toc179192199)

[2.11 Achieve Reliability Targets 13](#_Toc179192200)

[2.12 Test and Conduct Drills 13](#_Toc179192201)

[3 Security 13](#_Toc179192202)

[3.1 Design Principles 14](#_Toc179192203)

[3.2 Checklist 21](#_Toc179192204)

[3.3 Trade-offs 22](#_Toc179192205)

[3.4 Implement Defence in Depth 26](#_Toc179192206)

[3.5 Apply the Principle of Least Privilege 26](#_Toc179192207)

[3.6 Set Up Continuous Security Monitoring 26](#_Toc179192208)

[3.7 Conduct Regular Security Assessments 26](#_Toc179192209)

[4 Operational Excellence 26](#_Toc179192210)

[4.1 Design Principles 26](#_Toc179192211)

[4.2 Flow Optimisation 36](#_Toc179192212)

[4.3 Checklist 40](#_Toc179192213)

[4.4 Trade-offs 42](#_Toc179192214)

[4.5 Develop Robust Deployment Processes 44](#_Toc179192215)

[4.6 Implement Continuous Monitoring 44](#_Toc179192216)

[4.7 Maintain Detailed Documentation 44](#_Toc179192217)

[4.8 Conduct Regular Process Reviews 44](#_Toc179192218)

[5 Performance Efficiency 44](#_Toc179192219)

[5.1 Design Principles 45](#_Toc179192220)

[5.2 Checklist 49](#_Toc179192221)

[5.3 Trade-offs 50](#_Toc179192222)

[5.4 Design for Scalability 53](#_Toc179192223)

[5.5 Optimise System Performance 53](#_Toc179192224)

[5.6 Implement Effective Resource Management 53](#_Toc179192225)

[5.7 Conduct Regular Performance Testing 53](#_Toc179192226)

[6 Experience Optimisation 53](#_Toc179192227)

[6.1 Design Principles 53](#_Toc179192228)

[6.2 Checklist 57](#_Toc179192229)

[6.3 Trade-offs 58](#_Toc179192230)

[7 Cost Management 59](#_Toc179192231)

[7.1 Design Principles 59](#_Toc179192232)

[7.2 Checklist 65](#_Toc179192233)

[7.3 Trade-offs 67](#_Toc179192234)

[7.4 Implement Cost Tracking Mechanisms 71](#_Toc179192235)

[7.5 Optimise Resource Usage 71](#_Toc179192236)

[7.6 Set Cost Reduction Targets 71](#_Toc179192237)

[7.7 Review and Adjust Budgets Regularly 71](#_Toc179192238)

# Introduction

## Purpose

This document details the setting of standards, patterns and practices for development, build, quality assurance, and monitoring and reporting within Power Platform and D365 solutions.

This document will evolve as the platform evolves and this document will be expanded when necessary to capture any updates to process, technology or practice.

## Scope

The scope of this document is to establish standards, patterns and practices for the development, build, quality assurance, monitoring and reporting within Power Platform and D365 solutions.

## Areas of Scope

A diagram of a company

Description automatically generated

Based on the Power Platform Well-Architected Framework, the areas of scope within this document include:

* **Reliability**: Ensuring that solutions are reliable and can recover quickly from outages or failures.
* **Security**: Ensuring that all solutions adhere to stringent security protocols to protect data integrity and privacy.
* **Operational** **Excellence**: Developing robust processes for deployment, monitoring, and maintenance to ensure operational excellence.
* **Performance** **Efficiency**: Optimising resources to ensure that solutions perform efficiently and scale effectively.
* **Experience Optimisation**: Apply experience optimisation guidance to your architecture to provide meaningful and effective experiences for users of your workload.
* **Cost** **Management**: Implementing practices to manage and optimise costs associated with the development and deployment of solutions.

This comprehensive approach will ensure that our Power Platform and D365 solutions are not only effective but also sustainable and scalable.

# Reliability

A reliable workload must be resilient so that it can detect and recover from outages and malfunctions and consistently deliver functionality. It must be capable of recovering from failures within a reasonable timeframe. It must also be available so that users can consistently and reliably access the workload during the agreed timeframe and at the agreed quality level.

## Pillars

* **Resilience**: Design systems to be resilient to failures and to recover quickly.
* **Redundancy**: Implement redundancy at all critical points to avoid single points of failure.
* **Monitoring**: Develop comprehensive monitoring to detect and respond to issues promptly.

If you don't apply these principles to your design, the workload most likely won't be prepared to anticipate or handle problems in production. The outcome might be service disruptions that lead to financial loss. In the case of critical workloads, failing to apply these principles could jeopardise safety.

## Design for Business Requirements

**Collect and understand business needs with a focus on the expected use of the workload.**

Requirements should include the user experience, workflows, data, and workload-specific features. The goals should be realistic and agreed with all stakeholders, based on a given budget. The requirements should clearly define the expectations with the team and stakeholders. Document requirements to guide technology choices, implementations, and operations.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Quantify success by setting targets on indicators** for individual components, system flows, and the system. **Do those targets make user flows more reliable?** | Metrics quantify expectations. They enable you to **understand complexities** and determine whether the downstream costs of those complexities are within the investment limit.  The target values indicate an ideal state. You can use the values as test thresholds that help you **detect deviations** from that state and how long it takes to return to the target state.  Compliance requirements must also have predictable outcomes for in-scope flows. Prioritising these flows **bring attention to areas that are the most sensitive**. |
| Understand platform commitments. **Consider the limits, quotas, regions, and capacity constraints** for services. | Service-level agreements (SLAs) vary by service. Not all services and features are covered equally. Not all services or features are available in all regions. Most of the subscription resource limits are per region. Having a good **understanding of coverage and limits** can help you detect drift and build resiliency and recovery mechanisms. |
| **Determine dependencies** and their effect on resiliency. | Keeping track of dependent infrastructure, services, APIs, and functions developed by other teams or third parties helps you determine **whether the workload can operate in absence of those dependencies**. It also helps you **understand cascading failures** and **improve downstream operations**.  Developers can **implement resilient design patterns** to handle potential failures when you use external services that might be susceptible to failures. |

## Design for Resilience

**The workload must respond to failures and continue to operate with full or reduced functionality.**

Design your workload to be able to identify faults and component outages. Make the system resilient so that it can tolerate faults and handle them smoothly. Inform users about the system state and set the expectations of what components are unavailable and how long expected recovery will take.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Distinguish components that are on the critical path** from those that can function in a degraded state. | Not all components of the workload need to be equally reliable. Determining criticality helps you design according to the **criticality of each component**. You **won't overengineer resiliency for components** that could slightly deteriorate the user experience, as opposed to components that can cause end-to-end problems if they fail.  The design can be efficient in allocating resources to critical components. You can also implement fault isolation strategies so that if a noncritical component fails or enters a degraded state, it can be isolated to prevent cascading failures. |
| **Identify potential failure points in the system**, especially for the critical components, and determine the effect on user flows. | You can **analyse the failure cases, blast radius, and intensity of fault**: full or partial outage. This analysis influences the design of error handling capabilities at the component level. |
| **Build self-preservation capabilities** by using design patterns correctly and modularising the design to isolate faults. | The system will be able to **prevent a problem from affecting downstream components**. The system will be able to mitigate transient and permanent failures, performance bottlenecks, and other problems that might affect reliability.  You'll also be able to **minimise the blast radius**. |
| **Add the capability to scale out the critical components** (application and infrastructure) by considering the capacity constraints of services in the supported regions. | The workload will be able to **handle variable capacity spikes and fluctuations**. This capability is crucial when there's an unexpected load on the system, like a surge in valid usage. If the workload is designed to scale out over multiple regions it can even overcome potential temporary resource capacity constraints or other issues impacting in a single region. |
| **Build redundancy in layers and resiliency on various application tiers.**  Aim for redundancy in physical utilities and immediate data replication. Also aim for redundancy in the functional layer that covers services, operations, and personnel. | Redundancy helps **minimise single points of failure**. For example, if there’s a component, zonal, or regional outage, redundant deployment (in active-active or active-passive) allows you to meet uptime targets.  Adding intermediaries prevents direct dependency between components and improves buffering. Both benefits harden the resiliency of the system. |
| **Overprovision to immediately mitigate individual failure** of redundant instances and to buffer against runaway resource consumption. | Higher investment in overprovisioning **increases resiliency**.  The system will continue to operate at full utility during an active failure even before scaling operations can start to **remediate the failure**. Likewise, you can reduce the risk of unexpected runaway resource consumption claiming your planned buffer, gaining critical triage time, before system faults or aggressive scaling occurs. |

## Design for Recovery

**The workload must be able to recover from failures with minimal disruption to the user experience and business objectives.**

Put in place structured, tested, and documented recovery plans that lead to quick recovery. Ensure your support and operations teams have access to these plans and recovery drills are performed frequently to avoid confusion and chaos during an actual outage.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Have structured, tested, and documented recovery plans** that are aligned with the negotiated recovery targets. Plans must cover all components in addition to the system. | A well-defined process leads to a **quick recovery** that can prevent negative impact on the finances and reputation of your business. Conducting regular recovery drills tests the process of recovering system components, data, and failover and failback steps to **avoid confusion when time and data integrity are key** measures of success. |
| Ensure that you can **repair data** of all stateful components within your recovery targets. | Backups are essential to getting the **system back to a working state** by using a trusted recovery point, like the last-known good state.  Immutable and transactionally consistent backups ensure that data can't be altered, and that the restored data isn't corrupted. |
| Implement **automated self-healing capabilities** in the design. | This automation **reduces risks from external factors**, like human intervention, and shortens the break-fix cycle. |
| Replace stateless components with **immutable ephemeral units**. | Building ephemeral units that you can spin up and destroy on demand provides **repeatability and consistency**. Use side-by-side deployment models to make the transition to the new units incremental, minimising disruptions. |

## Design for Operations

**The workload must be observable, and development teams must be able to learn from failures.**

Build monitoring into your workload so that development and support teams have enough data to analyse when, why, and what component is failing. Build meaningful alerts to notify support teams in the event of a failure. Use analytics and insights to drive improvements.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Build observable systems** that can correlate telemetry. | Monitoring and diagnostics are crucial operations. If something fails, you need to know that it failed, **when it failed**, and **why it failed**. Observability at the component level is fundamental, but aggregated observability of components and correlated user flows provides a holistic view of health status. This data is required to enable site-reliability engineers to prioritise their efforts for remediation. |
| **Predict potential malfunctions and anomalous behaviour.** Make active reliability failures visible by using prioritised and actionable alerts.  Invest in reliable processes and infrastructure that leads to quicker triage. | Site reliability engineers can be notified immediately so that they can **mitigate ongoing live site incidents** and proactively mitigate potential failures identified by predictive alerts before they become live incidents. |
| **Simulate failures** and run tests in production and pre-production environments. | It's beneficial to experience failures in production so you can set **realistic expectations for recovery**. This allows you to make design choices that gracefully respond to failures. Also, it enables you to test the thresholds you set for business metrics. |
| Build components with **automation in mind** and automate as much as you can. | Automation **minimises the potential for human error**, bringing **consistency** to testing, deployment, and operations. |
| Factor in **routine operations and their impact** on the stability of the system. | The workload might be subject to ongoing operations, like application revisions, security and compliance audits, component upgrades, and backup processes. Scrutinising those changes ensures the **stability of the system**. |
| Continuously **learn from incidents in production**. | Based on the incidents, you can determine the impact and oversights in design and operations that might go unnoticed in preproduction. Ultimately, you'll be able to **drive improvements** based on real-life incidents. |

## Keep it Simple

**Avoid overengineering the architecture design, application code and operations.**

Focus on the business requirements and remove features and components that aren't necessary. Establish standards for development and deployment that increase consistency. Take advantage of platform-provided features and pre-built assets to minimise development time.

|  |  |
| --- | --- |
| Approach | Benefit |
| Add components to your architecture only if they help you achieve target business values. **Keep the critical path lean**. | Designing for business requirements can lead to a straightforward solution that's **easy to implement and manage**. Avoid having too many critical components, because each one is a significant point of failure. |
| **Establish standards** in code implementation, deployment, and processes, and document them. Identify opportunities to enforce those standards by using automated validations. | Standards provide **consistency and minimise human errors**. Approaches like standard naming conventions and code style guides can help you maintain quality and make assets easy to identify during troubleshooting. |
| Evaluate whether theoretical approaches translate to **pragmatic design** that applies to your use cases. | Application code that's too granular can lead to unnecessary interdependence, extra operations, and **difficult maintenance**. |
| **Develop just enough code**. | You'll be able to prevent problems that are the result of **inefficient implementations**, like unexpected resource consumption, user or dataflow failures, and code bugs.  Conversely, reliability problems should lead to code reviews to ensure that code is resilient enough to handle the problems. |
| **Take advantage of platform-provided features** and prebuilt assets that can help you effectively meet business targets. | This approach **minimises development time**. It also enables you to **rely on tried and tested practices** that have been used with similar workloads. |

## Checklist

Approach your design with a focus on reliability to help ensure that you design a workload that's resilient, manageable, and repeatable. If you don't include reliability practices and consider the trade-offs, your design is potentially at risk. Carefully consider all the points covered in the checklist to instil confidence in your system's success.

|  |  |
| --- | --- |
| Code | Recommendation |
| RE:01 | **Design your workload to align with business objectives and avoid unnecessary complexity or overhead**. Use a practical and balanced approach to make design decisions that deliver the desired results. Contain your design to the necessities to reduce inefficiencies and potential problems. |
| [RE:02](https://learn.microsoft.com/en-us/power-platform/well-architected/reliability/identify-flows) | **Identify and rate user and system flows.** Use a criticality scale based on your business requirements to prioritise the flows. |
| [RE:03](https://learn.microsoft.com/en-us/power-platform/well-architected/reliability/failure-mode-analysis) | **Use failure mode analysis (FMA) to identify and prioritise potential failures in your solution components.** Perform FMA to help you assess the risk and effect of each failure mode. Determine how the workload responds and recovers. |
| [RE:04](https://learn.microsoft.com/en-us/power-platform/well-architected/reliability/metrics) | **Define reliability and recovery targets** for the components, the flows, and the overall solution. Visualise the targets to **negotiate, gain consensus, set expectations, and drive actions** to achieve the ideal state. Use the defined targets to build the health model. The health model defines what healthy, degraded, and unhealthy states look like. |
| [RE:05](https://learn.microsoft.com/en-us/power-platform/well-architected/reliability/background-jobs) | **Strengthen the resiliency of your workload by implementing error handling and transient fault handling**. Build capabilities into the solution to handle component failures and transient errors. |
| [RE:06](https://learn.microsoft.com/en-us/power-platform/well-architected/reliability/testing-strategy) | **Test for resiliency and availability scenarios by applying the principles of chaos engineering in your test and production environments**. Use testing to ensure that your graceful degradation implementation strategies are effective by performing active malfunction and simulated load testing. |
| [RE:07](https://learn.microsoft.com/en-us/power-platform/well-architected/reliability/disaster-recovery) | **Implement structured, tested, and documented business continuity and disaster recovery (BCDR) plans that align with the recovery targets**. Plans must cover all components and the system. |
| [RE:08](https://learn.microsoft.com/en-us/power-platform/well-architected/reliability/monitoring-alerting-strategy) | **Measure and publish the solution's health indicators**. Continuously capture uptime and other reliability data from across the workload and from individual components and key flows. |

## Trade-offs

A reliable workload consistently meets its defined reliability objectives. It should reach established resiliency targets, ideally by circumventing events that affect reliability. Realistically, however, a workload must tolerate and control the impact of such events and maintain operations at a predetermined level during active malfunction. Even during a disaster, a reliable workload must recover to a specific state within a given period, both of which are agreed upon among the stakeholders. An incident response plan that enables you to achieve rapid detection and recovery is vital.

During the design phase of a workload, you need to consider how decisions based on the Reliability design principles and the recommendations in the Design review checklist for Reliability might influence the goals and optimisations of other pillars. Certain decisions might benefit some pillars but constitute a trade-off for others. This section describes example trade-offs that a workload team might encounter when designing workload architecture and operations for reliability.

**Reliability trade-offs with Security**

**Increased workload surface area**. The Security pillar prioritises a reduced and contained surface area to minimise attack vectors and reduce the management of security controls.

* Reliability is often obtained through replication. Replication can occur at the component level, at the data level, or even at a geographic level. Replicas, by design, increase the surface area of a workload. From a security perspective, a reduced and contained surface area is preferred to minimise potential attack vectors and streamline the management of security controls.
* Similarly, disaster recovery solutions, like backups, increase a workload's surface area. However, they're often isolated from the workload's runtime. This requires the implementation of additional security controls, which might be specific to the disaster recovery solution.
* For the sake of reliability goals, additional components might be needed for the architecture, which increases the surface area. This increased complexity increases the surface area of the workload by adding new components that need to be secured, possibly in ways that aren't already used in the system. Typically, these components are accompanied by additional code to support their use or general reliability patterns, which also increases the application's surface area.

**Security control bypass**. The Security pillar recommends that all controls remain active in both normal and stressed systems.

* When a workload is experiencing a reliability event that's being addressed under active incident response, urgency might create pressure for workload teams to bypass security controls that are optimised for routine access.
* Troubleshooting activities can cause the team to temporarily disable security protocols, leaving an already stressed system potentially exposed to additional security risks. There's also a risk that the security protocols won't be reestablished promptly.
* Granular implementations of security controls, like role-based access control assignments or firewall rules, introduce configuration complexity and sensitivity, increasing the chance for misconfiguration. Mitigating this potential reliability impact by using broad rules erodes all three Zero Trust architecture principles.

**Old software versions**. The Security pillar encourages a "get current, stay current" approach to vendor security patches.

* Applying release wave updates or updates to vendor libraries, like third-party components or solutions, can potentially disrupt the target component, causing unavailability during the change. Delaying or avoiding patching might avoid the potential reliability risks, but it leaves the system unprotected against evolving threats.
* The preceding consideration also applies to the workload's code. For example, it applies to application code that uses old libraries and components. If updating and deploying application code is viewed as an unmitigated reliability risk, the application is exposed to additional security risks over time.

**Reliability trade-offs with Operational Excellence**

**Increased operational complexity**. Operational Excellence, like Reliability itself, prioritises simplicity.

* Having a comprehensive monitoring strategy for a workload is a key part of operational excellence. Introducing additional components into an architecture to implement reliability design patterns results in more data sources to manage, increasing the complexity of implementing distributed tracing and observability.
* Using multiple regions to overcome single region resource capacity constraints and/or implement an active/active architecture increases the complexity of the workload's operational management. This complexity is introduced by the need to manage multiple regions and the need to manage the data replication between them.

**Increased effort to generate team knowledge and awareness**. The Operational Excellence pillar recommends keeping and maintaining a documentation repository for procedures and topologies.

* As a workload becomes more robust through the addition of reliability components and patterns, it takes more time to maintain operational procedures and artifact documentation.
* Training becomes more complex as the number of components in the workload increases. This complexity affects the time required for onboarding and increases the knowledge that's needed to track product roadmaps and service-level guidance.

**Reliability trade-offs with Experience Optimisation**

**The Experience optimisation pillar prioritises user efficiency.**

* Emphasising rigorous testing can delay the release of experience features that are essential for adoption.
* Optimising for reliability can over index on minimising complexity, which deprioritises features for more engaging user experiences, such as custom components and integrations.

**Reliability trade-offs with Performance Efficiency**

**Performance Efficiency requires a system to achieve performance targets for user and data flows.**

* Reliability patterns often incorporate data replication to survive replica malfunction. Replication introduces additional latency for reliable data-write operations, which consumes a part of the performance budget for a specific user or data flow.
* Reliability sometimes employs various forms of resource balancing to distribute or redistribute load to healthy replicas. A dedicated component that's used for balancing usually affects the performance of the request or process that's being balanced.
* Distributing components across geographical boundaries or availability zones to survive a scoped impact introduces network latency in the communication between components that span those availability boundaries.
* Extensive processes are used to observe the health of a workload. Although monitoring is critical for reliability, instrumentation can affect system performance. As observability increases, performance might decrease.

**The Performance Efficiency pillar discourages over-provisioning, instead recommending the use of just enough resources to satisfy demand.**

* Automatic scaling operations aren't instantaneous and therefore can't reliably handle a sudden and dramatic spike in demand that can't be shaped or smoothed. Therefore, over-provisioning via either larger instances or more instances is a critical reliability tactic to account for the lag between demand signal and supply creation. Unused capacity counters the goals of performance efficiency.
* Sometimes a component can't be scaled in reaction to demand, and that demand isn't fully predictable. Using large instances to cover the worst case leads to over-provisioning waste in situations that are outside that use case.

## Review Design Principles

Regularly review and update design principles to ensure they align with the latest industry standards and best practices. Conduct workshops and feedback sessions to gather input from stakeholders.

## Set and Measure Reliability Targets

Define clear reliability targets, such as uptime percentages and mean time to recovery (MTTR). Use monitoring tools to measure these targets and generate reports to track performance over time.

## Achieve Reliability Targets

Regularly test systems against predefined reliability targets using simulations and stress tests. Implement continuous improvement processes to address any shortcomings.

## Test and Conduct Drills

Conduct regular disaster recovery drills to ensure all teams are prepared for real-world failures. Test failover mechanisms and recovery procedures under different scenarios to identify potential weaknesses.

# Security

A Well-Architected workload must be built with a zero-trust approach to security. A secure workload is resilient to attacks and incorporates the interrelated security principles of confidentiality, integrity, and availability (also known as the *CIA triad*) in addition to meeting business goals. Any security incident has the potential to become a major breach that damages your brand and reputation. To assess how well your security strategy works for your workload, ask yourself these questions:

* How much do your security measures slow down or stop attackers from breaking into your workload?
* How much do your security measures limit the damage or spread of an attack if it happens?
* How valuable is your workload to an attacker? How much would it hurt your business if your workload or its data were stolen, unavailable, or tampered with?
* How quickly can you detect, respond to, and recover from disruptions in your workload?

**As you design your system, use the Microsoft Zero Trust model as the compass to mitigate security risks:**

* Verify explicitly so that only trusted identities perform intended and allowed actions that originate from expected locations. This safeguard makes it harder for attackers to impersonate legitimate users and accounts.
* Use least-privilege access for the right identities, with the right set of permissions, for the right duration, and to the right assets. Limiting permissions helps keep attackers from abusing permissions that legitimate users don't even need.
* Assume breach of security controls and design compensating controls that limit risk and damage if a primary layer of defence fails. Doing so helps you to defend your workload better by thinking like an attacker who's interested in success (regardless of how they get it).

Security isn't a one-time effort. You must implement this guidance on a recurring basis. Continuously improve your defences and security knowledge to protect your workload from attackers who are adept at finding new, innovative attack vectors, often using automated attack kits.

The design principles, based on the Microsoft Azure Well-Architected Framework, are intended to foster a continuous security mindset, to help you improve the security posture of your workload as threats evolve. These principles should guide the security of your architecture, design choices, and operational processes. Start with the recommended approaches and justify the benefits for a set of security requirements. After you set your strategy, drive actions by using the Security checklist as your next step.

If these principles aren't applied properly, a negative impact on business operations and revenue can be expected. Some consequences might be obvious, like penalties for regulatory workloads. However, others may be less apparent and could result in ongoing security problems before they're detected.

In many mission-critical workloads, security is the primary concern, alongside reliability, given that some attack vectors, like data exfiltration, don't affect reliability. Security and reliability can pull a workload in opposite directions because security-focused design can introduce points of failure and increase operational complexity. The effect of security on reliability is often indirect, introduced by way of operational constraints. Carefully consider trade-offs between security and reliability.

## Design Principles

* **Defence in Depth**: Implement multiple layers of security controls to protect data and systems.
* **Least Privilege**: Grant the minimum level of access required for users and applications.
* **Continuous Monitoring**: Continuously monitor security threats and take proactive measures to mitigate risks.

By following these principles, you can improve security effectiveness, harden workload assets, and build trust with your users.

**Plan Your Security Readiness**

**Aim to adopt and implement security practices in architectural design decisions and operations with minimal friction.**

As a workload owner, you share the responsibility with the organisation to protect assets. Make a security readiness plan that matches your business priorities. It will assist you in establishing clear processes, sufficient investments, and appropriate responsibilities. The plan should communicate the workload requirements to the organisation, which also shares responsibility for protecting assets. Security plans should be part of your strategy for reliability, health modelling, and self-preservation.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Use segmentation as a strategy to plan security boundaries** in the workload environment, processes, and team structure to **isolate access and function**.  Your segmentation strategy should be driven by business requirements. You can base it on criticality of components, division of labor, privacy concerns, and other factors. | You'll be able to **minimise operational friction** by defining roles and establishing **clear lines of responsibility**. This exercise also helps you **identify the level of access** for each role, especially for critical-impact accounts.  Isolation enables you to **limit exposure of sensitive flows** to only roles and assets that need access. Excessive exposure could inadvertently lead to information flow disclosure.  To summarise, you'll be able to **right-size security efforts** based on the needs of each segment. |
| Continuously **build skills** through **role-based security training** that meets the requirements of the organisation and the use cases of the workload. | A highly skilled team can design, implement, and monitor **security controls that remain effective** against attackers, who constantly look for new ways to exploit the system.  Organisation-wide training typically focuses on developing a broader skill set for securing the common elements. However, with role-based training, you focus on **developing deep expertise** in the platform offerings and security features that address workload concerns.  You need to implement both approaches to defend against adversaries through **good design and effective operations**. |
| **Make sure there's an incident response plan** for your workload.  Use industry frameworks that define the standard operating procedure for preparedness, detection, containment, mitigation, and post-incident activity. | At the time of crisis, confusion must be avoided.  If you have a well-documented plan, responsible roles can **focus on execution** without wasting time on uncertain actions. Also, a comprehensive plan can help you ensure that **all remediation requirements are fulfilled**. |
| **Strengthen your security posture by understanding the security compliance requirements** that are imposed by influences outside the workload team, like organisational policies, regulatory compliance, and industry standards. | Clarity about compliance requirements will help you **design for the right security assurances** and **prevent non-compliance** issues, which could lead to penalties.  Industry standards can provide a baseline and influence your choice of tools, policies, security safeguards, guidelines, risk-management approaches, and training.  If you know that the workload adheres to compliance, you'll be able to **instill confidence** in your user base. |
| **Define and enforce team-level security standards** across the lifecycle and operations of the workload.  **Strive for consistent practices** in operations like coding, gated approvals, release management, and data protection and retention. | Defining good security practices can **minimise negligence** and the surface area for potential errors. The team will **optimise efforts and the outcome will be predictable** because approaches are made more consistent.  Observing security standards over time will enable you to **identify opportunities for improvement, possibly including automation**, which will streamline efforts further and increase consistency. |
| Align your incident response with the **Security Operation Center (SOC) Centralised function** in your organisation. | Centralising incident response functions enables you to take advantage of specialised IT professionals who can detect incidents in real time to address potential threats as quickly as possible. |

**Design To Protect Confidentiality**

**Prevent exposure of privacy, regulatory, application, and proprietary information by using access restrictions and obfuscation techniques.**

Workload data can be classified by user, usage, configuration, compliance, intellectual property, and more. You shouldn't share or access that data beyond the established trust boundaries. To protect confidentiality, you should focus on access controls, opacity, and keeping an audit trail of activities that involve data and the system.

|  |  |
| --- | --- |
| Approach | Benefit |
| Implement **strong access controls** that grant access only on a need-to-know basis. | *Least privilege*.  The workload will be protected from **unauthorised access** and prohibited activities. Even when access is from trusted identities, the **access permissions and exposure time will be minimised** because the communication path is open for a limited period. |
| **Classify data based on its type, sensitivity, and potential risk**. Assign a confidentiality level for each.  Include system components that are in scope for the identified level. | *Verify explicitly*.  This evaluation helps you right-size security measures.  You'll also be able to identify data and components that have a **high potential impact** and/or exposure to risk. This exercise adds **clarity** to your information protection strategy and helps ensure **agreement**. |
| Safeguard your data at rest, in transit, and during processing by using **encryption**. Base your strategy on the assigned confidentiality level. | *Assume breach*.  Even if an attacker gets access, they **won't be able to read properly encrypted** sensitive data.  Sensitive data includes configuration information that's used to gain further access inside the system. Data encryption can help you **contain risks**. |
| **Guard against exploits** that might cause unwarranted exposure of information. | *Verify explicitly*.  It's crucial to minimise vulnerabilities in authentication and authorisation implementations, code, configurations, operations, and those that stem from the social habits of the system's users.  Up-to-date security measures enable you to **block known security vulnerabilities** from entering the system. You can also **mitigate new vulnerabilities** that can appear over time by implementing routine operations throughout the development cycle, continuously improving security assurances. |
| **Guard against data exfiltration** that results from malicious or inadvertent access to data. | *Assume breach*.  You'll be able to contain blast radius by **blocking unauthorised data transfer**. Additionally, controls applied to networking, identity, and encryption will protect data at various layers. |
| **Maintain the level of confidentiality** as data flows through various components of the system. | *Assume breach*.  Enforcing confidentiality levels throughout the system enables you to provide a consistent level of hardening. Doing so can **prevent vulnerabilities** that might result from moving data to a lower security tier. |
| Maintain an **audit trail** of all types of access activities. | *Assume breach*.  Audit logs support **faster detection and recovery** in case of incidents and help with ongoing security monitoring. |

**Design To Protect Integrity**

**Avoid damage to design, implementation, operations, and data to prevent disruptions that can stop the system from delivering its expected value or cause it to operate outside the defined limits. The system should provide information assurance throughout the workload lifecycle.**

The key is to use controls that prevent tampering of business logic, flows, deployment processes, data, and even the lower stack components, like the operating system and boot sequence. Lack of integrity can create vulnerabilities that can lead to breaches in confidentiality and availability.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Implement strong access controls that authenticate and authorise access to the system.**  **Minimise access based on privilege, scope, and time.** | *Least privilege*.  Depending on the strength of the controls, you'll be able to **prevent or reduce risks from unapproved modifications**. This helps ensure that data is consistent and trustworthy.  Minimising access limits the extent of potential corruption. |
| **Continuously protect against vulnerabilities and detect them in your supply chain** to block attackers from injecting software faults into your infrastructure, build system, tools, libraries, and other dependencies.  Supply chain should scan for vulnerabilities during **build time and runtime**. | *Assume breach*.  Knowing the origin of software and verifying its authenticity throughout the lifecycle will **provide predictability**. You'll **know about vulnerabilities well in advance** so that you can proactively remediate them and keep the system secure in production. |
| **Establish trust and verify by using cryptography techniques** like attestation, code signing, certificates, and encryption.  Protect those mechanisms by allowing reputable decryption. | *Verify explicitly*, *least privilege.*  You'll know that changes to data or access to the system **is verified by a trusted source**.  Even if encrypted data is intercepted in transit by a malicious actor, the actor won't be able to unlock or decipher the content. You can use digital signatures to ensure that the data wasn't tampered with during transmission. |
| **Ensure backup data is immutable and encrypted** when data is replicated or transferred. | *Verify explicitly.*  You'll be able to recover data with confidence that backup **data wasn't changed at rest**, inadvertently or maliciously. |
| **Avoid or mitigate system implementations that allow your workload to operate outside its intended limits and purposes.** | *Verify explicitly.*  When your system has strong safeguards that check whether usage aligns with its intended limits and purposes, the scope for potential abuse or tampering of your compute, networking, and data stores is reduced. |

**Design To Protect Availability**

**Avoid or minimise system and workload downtime and degradation in the event of a security incident by using strong security controls. You must keep data integrity during the incident and after the system recovers.**

You must balance availability architecture choices with security architecture choices. The system should provide availability guarantees to ensure that users can access data, and that the data is reachable. From a security perspective, users should operate within the allowed access scope, and the data must be trusted. Security controls should stop bad actors, but they shouldn't stop legitimate users from accessing the system and data.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Prevent compromised identities from misusing access** to gain control of the system.  Check for **overly pervasive scope and time limits** to minimise risk exposure. | *Least privilege*.  This strategy **mitigates the risks of excessive, unnecessary, or misused access permissions** on crucial resources. Risks include unauthorised modifications and even the deletion of resources. Take advantage of the platform-provided just-in-time (JIT), just-enough-access (JEA), and time-based security modes to replace standing permissions wherever possible. |
| Use security controls and design patterns to **prevent attacks and code flaws from causing resource exhaustion** and blocking access. | *Verify explicitly*.  The **system won't experience downtime** caused by malicious actions, like distributed denial of service (DDoS) attacks. |
| Implement **preventative measures for attack vectors that exploit vulnerabilities** in application code, networking protocols, identity systems, malware protection, and other areas. | *Assume breach*.  Implement code scanners, apply the latest security patches, update software, and protect your system with effective antimalware on an ongoing basis.  You'll be able to reduce the attack surface to ensure business continuity. |
| **Prioritise** security controls on the **critical components and flows** in the system that are susceptible to risk. | *Assume breach*, *verify explicitly*.  Regular detection and prioritisation exercises can help you **apply security expertise to the critical aspects** of the system. You'll be able to focus on the most likely and damaging threats and start your risk mitigation in areas that need the most attention. |
| Apply at least the same level of **security rigor in your recovery resources and processes** as you do in the primary environment, including security controls and frequency of backup. | *Assume breach*.  You should have a preserved safe system state available in disaster recovery. If you do, you can fail over to a secure secondary system or location and restore backups that won't introduce a threat.  A well-designed process can prevent a security incident from hindering the recovery process. Corrupted backup data or encrypted data that can't be deciphered can slow down recovery. |

**Sustain And Evolve Your Security Posture**

**Include continuous improvement and apply vigilance to stay ahead of attackers who are continuously evolving their attack strategies.**

Your security posture must not deteriorate over time. You must keep improving security operations so that new disruptions are handled more effectively. Aim to align improvements with the phases defined by industry standards. Doing so leads to better readiness, faster time to incident detection, and effective containment and mitigation. Continuous improvement should be based on lessons learned from past incidents.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Create and maintain a comprehensive asset inventory** that includes classified information about resources, locations, dependencies, owners, and other metadata that's relevant to security.  As much as possible, **automate** inventory to derive data from the system. | A well-organised inventory provides a **holistic view of the environment**, which puts you in an advantageous position against attackers, especially during post-incident activities.  It also creates a business rhythm to drive communication, upkeep of critical components, and the decommissioning of orphaned resources. |
| **Perform threat modeling** to identify and mitigate potential threats. | You'll have a **report of attack vectors** prioritised by their severity level. You'll be able to identify threats and vulnerabilities quickly and set up countermeasures. |
| Regularly **capture data to quantify your current state** against your established security baseline and **set priorities for remediations**.  Take advantage of platform-provided features for **security posture management** and **the enforcement of compliance** imposed by external and internal organisations. | You need accurate reports that bring clarity and consensus to focus areas. You'll be able to immediately **execute technical remediations**, starting with the highest priority items. You'll also **identify gaps**, which provide opportunities for improvement.  Implementing enforcement helps prevent violations and regressions, which preserves your security posture. |
| **Run periodic security tests** that are conducted by experts external to the workload team who attempt to ethically hack the system.  Perform routine and integrated **vulnerability scanning** to detect exploits in infrastructure, dependencies, and application code. | These tests enable you to validate security defenses by **simulating real-world attacks** by using techniques like penetration testing.  Threats can be introduced as part of your change management. Integrating scanners into the deployment pipelines enables you to automatically detect vulnerabilities and even quarantine usage until the vulnerabilities are removed. |
| **Detect, respond, and recover** with swift and effective security operations. | The primary benefit of this approach is that it enables you to **preserve or restore the security assurances of the CIA triad** during and after an attack.  You need to be alerted as soon as a threat is detected so that you can start your investigations and take appropriate actions. |
| **Conduct post-incident activities** like root-cause analyses, postmortems, and incident reports. | These activities provide insight into the impact of the breach and into resolution measures, which drives improvements in defenses and operations. |
| **Get current, and stay current.**  Stay current on updates, patching, and security fixes.  Continuously evaluate the system and improve it based on audit reports, benchmarking, and lessons from testing activities. Consider automation, as appropriate.  Use threat intelligence powered by security analytics for dynamic detection of threats.  At regular intervals, review the workload's conformance to Security Development Lifecycle (SDL) best practices. | You'll be able to ensure that your **security posture doesn't degrade over time**.  By integrating findings from real-world attacks and testing activities, you'll be able to combat attackers who continuously improve and exploit new categories of vulnerabilities.  Automation of repetitive tasks **decreases the chance of human error** that can create risk.  SDL reviews bring clarity around security features. SDL can help you maintain an inventory of workload assets and their security reports, which cover origin, usage, operational weaknesses, and other factors. |

## Checklist

* **Implement Defence in depth.**
* **Apply the principle of least privilege.**
* **Set up continuous security monitoring.**
* **Conduct regular security assessments.**

This checklist presents a set of security recommendations to help you ensure your workload is secure. If you don't review the checklist and weigh the associated trade-offs, you may expose your design to potential risks. Thoroughly assess all the aspects outlined in the checklist to enhance your confidence in the security of your workload.

|  |  |
| --- | --- |
| Code | Recommendation |
| SE:01 | **Establish a security baseline** that's aligned to compliance requirements, industry standards, and platform recommendations. Regularly measure your workload architecture and operations against the baseline to sustain or improve your security posture over time. |
| [SE:02](https://learn.microsoft.com/en-us/power-platform/well-architected/security/secure-development-lifecycle) | **Maintain a secure development lifecycle** by using a hardened, mostly automated, and auditable software supply chain. Incorporate a secure design by using threat modelling to safeguard against security-defeating implementations. |
| [SE:03](https://learn.microsoft.com/en-us/power-platform/well-architected/security/data-classification) | **Classify and consistently apply sensitivity and information type labels** on all workload data and systems involved in data processing. Use classification to influence workload design, implementation, and security prioritisation. |
| [SE:04](https://learn.microsoft.com/en-us/power-platform/well-architected/security/segmentation) | **Create intentional segmentation and perimeters** in your architecture design and in the workload's footprint on the platform. The segmentation strategy must include networks, roles and responsibilities, workload identities, and resource organisation. |
| [SE:05](https://learn.microsoft.com/en-us/power-platform/well-architected/security/identity-access) | **Implement strict, conditional, and auditable identity and access management (IAM)** across all workload users, team members, and system components. Limit access exclusively to *as necessary*. Use modern industry standards for all authentication and authorisation implementations. Restrict and rigorously audit access that's not based on identity. |
| [SE:06](https://learn.microsoft.com/en-us/power-platform/well-architected/security/encryption) | **Encrypt data by using modern, industry-standard methods** to guard confidentiality and integrity. Align the encryption scope with data classifications and prioritise native platform encryption methods. |
| [SE:07](https://learn.microsoft.com/en-us/power-platform/well-architected/security/application-secrets) | **Protect application secrets** by hardening their storage and restricting access and manipulation and by auditing those actions. Run a reliable and regular rotation process that can improvise rotations for emergencies. |
| [SE:08](https://learn.microsoft.com/en-us/power-platform/well-architected/security/monitor-threats) | **Implement a holistic monitoring strategy** that relies on modern threat detection mechanisms that can be integrated with the platform. Mechanisms should reliably alert for triage and send signals into existing SecOps processes. |
| [SE:09](https://learn.microsoft.com/en-us/power-platform/well-architected/security/testing) | **Establish a comprehensive testing regimen** that combines approaches to prevent security issues, validate threat prevention implementations, and test threat detection mechanisms. |
| [SE:10](https://learn.microsoft.com/en-us/power-platform/well-architected/security/incident-response) | **Define and test effective incident response procedures** that cover a spectrum of incidents, from localised issues to disaster recovery. Clearly define which team or individual runs a procedure. |

## Trade-offs

* **Security vs. Usability**: Enhanced security measures may impact system usability and user experience.
* **Cost vs. Security**: Implementing comprehensive security controls can be costly.

Security provides confidentiality, integrity, and availability assurances of a workload's system and its users' data. Security controls are required for the workload and for the software development and operational components of the system. When teams design and operate a workload, they can almost never compromise on security controls.

During the design phase of a workload, it's important to consider how decisions based on the Security design principles and recommendations in the Design review checklist for Security might influence the goals and optimisation efforts of other pillars. Certain decisions may benefit some pillars, while being trade-offs for others. This section lists example trade-offs that a workload team might encounter when designing workload architecture and operations for experience optimisation.

**Security trade-offs with Reliability**

The Reliability pillar prioritises simplicity and recommends that points of failure are minimised.

* Some security controls can increase the risk of misconfiguration, which can lead to service disruption. Examples of security controls that can introduce misconfiguration include network traffic rules, IP firewall settings, and role-based or attribute-based access control assignments.
* Workload security tooling is often incorporated into many layers of a workload's architecture, operations, and runtime requirements. These tools might affect resiliency, availability, and capacity planning. Failure to account for limitations in the tooling can lead to a reliability event, like Source Network Address Translation (SNAT) port exhaustion on an egress firewall.

The Reliability pillar recommends minimising critical dependencies. A workload that minimises critical dependencies, especially external ones, has more control over its points of failure.

The Security pillar requires a workload to explicitly verify identities and actions. Verification occurs via critical dependencies on key security components. If those components aren't available or if they malfunction, verification might not complete. This failure puts the workload in a degraded state. Some examples of these critical single-point-of-failure dependencies are:

* Ingress and egress firewalls
* Certificate revocation lists
* Identity providers, like Microsoft Entra ID

A workload must reliably recover from all forms of disaster.

* Security controls might affect recovery time objectives. This effect can be caused by the additional steps that are needed to decrypt backed up data or by operational access delays created by site reliability triage.
* Security controls themselves, for example secret vaults and their contents, need to be part of the disaster recovery plan of the workload and must be validated via recovery drills.
* Security or compliance requirements might limit data residency options or access control restrictions for backups, potentially further complicating recovery.

A workload that experiences runtime change is exposed to more risk of reliability impact due to that change.

* Stricter solution update policies lead to more changes in a workload's production environment. This change comes from sources like these:
  + Releasing application code more frequently because of updates to solutions
  + Applying Power Platform release wave updates
  + Updating configurations of Power Platform environment settings in the environment
  + Applying patches to libraries or components used in the environment
* Rotation activities for keys, service principal credentials, and certificates increase the risk of transient issues due to the timing of the rotation and clients using the new value.

**Security trade-offs with Operational Excellence**

Operational Excellence requires architectures to be serviceable and observable. The most serviceable architectures are those that are the most transparent to everyone involved.

* Security benefits from extensive logging that provides high fidelity insight into the workload for alerting on deviations from baselines and for incident response. This logging can generate a significant volume of logs, which can make it harder to provide insights that are targeted at reliability or performance.
* When compliance guidelines for data masking are followed, specific segments of logs or even large amounts of tabular data are redacted to protect confidentiality. The team needs to evaluate how this observability gap might affect alerting or hinder incident response.
* Some security controls impede access by design. During incident response, these controls can slow down workload operators' emergency access. Therefore, incident response plans need to include more emphasis on planning and drills in order to reach acceptable efficacy.

**Workload teams measure their velocity so that they can improve the quality, frequency, and efficiency of delivery activities over time. Workload complexity factors into the effort and risk involved in operations.**

* Stricter change control and approval policies to reduce the risk of introducing security vulnerabilities can slow down the development and safe deployment of new features. However, the expectation of addressing security updates and patching can increase demand for more frequent deployments. Additionally, human-gated approval policies in operational processes can make it more difficult to automate those processes.
* Security testing results in findings that need to be prioritised, potentially blocking planned work.
* Routine, ad hoc, and emergency processes might require audit logging to meet compliance requirements. This logging increases the rigidity of running the processes.
* Workload teams might increase the complexity of identity management activities as the granularity of role definitions and assignments is increased.
* An increased number of routine operational tasks that are associated with security, like certificate management, increases the number of processes to automate.

**A team that minimises external points of contact and review can control their operations and timeline more effectively.**

* As external compliance requirements from the larger organisation or from external entities increase, the complexity of achieving and proving compliance with auditors also increases.
* Security requires specialised skills that workload teams don't typically have. Those proficiencies are often sourced from the larger organisation or from third parties. In both cases, coordination of effort, access, and responsibility needs to be established.
* Compliance or organisational requirements often require maintained communication plans for responsible disclosure of breaches. These plans must be factored into security coordination efforts

**Security trade-offs with Experience Optimisation**

Optimising the experience focuses on helping users be more productive and make faster decisions.

* Security surface areas should be minimised, which can negatively impact the use of third-party components and integrations that are desired for optimising the experience.
* Data classification can make finding and consuming data in the workload more difficult.
* Security protocols increase the complexity of user interactions and can result in challenges for usability.

**Security trade-offs with Performance Efficiency**

A performant workload reduces latency and overhead.

* Inspection security controls, like firewalls and content filters, are located in the flows that they secure. Those flows are therefore subject to additional verification, which adds latency to requests.
* Identity controls require each invocation of a controlled component to be verified explicitly. This verification consumes compute cycles and might require network traversal for authorisation .
* Encryption and decryption require dedicated compute cycles. These cycles increase the time and resources consumed by those flows. This increase is usually correlated with the complexity of the algorithm and the generation of high-entropy and diverse initialisation vectors (IVs).
* As the extensiveness of logging increases, the impact on system resources and network bandwidth for streaming those logs can also increase.
* Resource segmentation frequently introduces network hops in a workload's architecture.

Reliably meeting performance targets depends on predictable implementations of the design.

A misconfiguration or overextension of security controls can impact performance because of inefficient configuration. Examples of security control configurations that can affect performance include:

* Firewall rule ordering, complexity, and quantity (granularity).
* Failing to exclude key files from file integrity monitors or virus scanners. Neglecting this step can lead to lock contention.
* Web application firewalls performing deep packet inspection for languages or platforms that are irrelevant for the components that are being protected.

## Implement Defence in Depth

Use multiple layers of security controls, including firewalls, encryption, and intrusion detection systems. Ensure that each layer addresses different aspects of security.

## Apply the Principle of Least Privilege

Grant users and applications the minimum level of access required to perform their functions. Regularly review and update access controls to maintain security.

## Set Up Continuous Security Monitoring

Implement security monitoring tools to detect and respond to threats in real-time. Use automated alerts to notify security teams of potential issues.

## Conduct Regular Security Assessments

Perform regular security assessments, including vulnerability scans and penetration testing. Use assessment results to identify and address security weaknesses.

# Operational Excellence

The Operational Excellence pillar defines processes for development practices, monitoring, and release management. The goal is to establish standards that reduce development time, human error, and user disruption. By following fusion development practices, your team will also collaborate better.

To assess your operational health, start with these questions:

* How do you drive continuous improvement and learn from experience?
* Do you have development and deployment standards that reduce friction and drive consistency?
* Is your workload meeting user expectations and requirements?

Without standards and clear leadership, workload teams often resort to methods that don't follow best practices, which can lead to poor user and support experiences.

## Design Principles

* **Automation**: Automate deployment, monitoring, and maintenance tasks to reduce human error.
* **Documentation**: Maintain comprehensive documentation for all processes and procedures.
* **Continuous** **Improvement**: Implement feedback loops to continuously improve operational processes.

These design principles, based on the Operational Excellence principles in the Microsoft Azure Well-Architected Framework, provide guidelines for operational strategies aimed at addressing root causes rather than merely treating symptoms. Start with the recommended approaches, and then observe what works and what doesn't to identify areas of improvement. After you set your strategy, continue to drive action by using the Operational Excellence checklist.

Your workload needs to meet both operational and business requirements. Efficient processes help the workload reach business goals while following compliance and governance standards. The key is to find repeatability with consistency.

If you meet these goals, workloads run reliably and predictably even during times of change. Not focusing on operational requirements might lead to failed and inconsistent deployments, added cost and time, and disrupted user experience.

**Embrace A Fusion Development and DevOps Culture**

**Empower development and operations teams to continuously improve the workload and processes by working together with a mindset of collaboration, shared responsibility, and ownership.**

Fusion development, or fusion teams’ development, can be viewed as distributed and multidisciplinary digital business teams that blend technology and other types of domain expertise.

Fusion development is a strategy aimed at scaling application development by using low-code methodologies. Fusion development allows your business to build better applications faster, by bringing together professional developers with citizen, or low-code, developers. It can also include using low-code capabilities and combining them with code-first components to meet business needs and create fusion applications.

A good fusion development and DevOps culture fosters a collaborative environment of shared knowledge and thrives on shared responsibility. Development and operations teams who are aligned on goals and priorities keep business focus and goals in mind. Feedback is shared among all teams. For example, operations teams share feedback on supporting the workload with the development team to improve the workload over time. At the same time, IT professionals and operations teams establish a secure, governed environment that empowers citizen developers to achieve more with minimum friction.

DevOps practices assign clear roles and responsibilities to each team and streamline operational tasks so that they're efficient but not overwhelming. To maximise the value of DevOps, the culture should improve processes with technology and promote open communication within the organisation.

|  |  |
| --- | --- |
| Approach | Benefits |
| **Use common systems and tools** that promote a collaborative environment for communication and tracking progress. | Common tools and processes enable **transparent communication**. Both development and operations teams benefit from situational awareness across various environments, common support issues, and overall challenges and wins.  Teams will already be familiar with existing escalation paths if there's an incident.  A shared backlog makes priorities, such as working on new features or fixing bugs, clear. |
| Build a **continuous learning and experimentation mindset** throughout the development cycle.  **Support knowledge sharing** across teams and maintain documentation for reuse.  Conduct blameless analysis and debrief post-release and/or post-incident reviews. | Through experimentation mechanisms, such as A/B testing and developing proofs of concept, you can encourage innovation while keeping costs low.  Share knowledge through collaboration that makes the team proficient in design approaches, tooling, and processes.  Doing retrospectives after a project helps **identify areas for improvement** and celebrate success. |
| **Adopt proven industry agile practices** that focus on action optimisation.  Look for **opportunities to "shift left"** in operations for manual and automated processes, deployment and quality assurance practices, and observability. | Agile development practices lead to shorter release lifecycles, which are an indicator of business value.  Detecting, resolving, and thereby preventing issues earlier is often less intrusive to the process. |
| **Set standards for all development and operational procedures and review and validate them at a regular cadence**.  These procedures include routine tasks, out-of-band processes, emergency drills and situations, choice of tooling, monitoring procedures, skilling plans, and even communication with stakeholders and customer disclosures.  **Be intentional and explicit about your decisions**. | Standards add predictability to operations and make processes and practices scalable. Validating standards is a great way to draw points of improvement.  Be prepared for emergency and recovery situations by conducting regular drills.  Execute with precision and enable governance to **prevent anomalies** that lead to risks. |
| **Take advantage of centralised operations teams** with **specialised skills** and breadth of experience. | There's a cost benefit to using shared resources both for operations and resources.  Although you own your workload, the centralised team helps you with cross-functional skills, such as incident management, a proactive perspective on monitoring, and outsourcing expertise with trust. |

**Establish Development Standards**

**Optimise productivity by standardising development practices, enforcing quality gates, and tracking progress and success through systematic change management.**

Technology and coding standards, style guides, and tools are in place to drive consistency and easier maintenance. Quality assurance processes ensure that functional and nonfunctional requirements are met and Emphasise testing early in the development life cycle. Effective processes are in place to Standardise technical activities and drive consensus within the team and stakeholders. Developers are supported by standards and process but not burdened by them.

|  |  |
| --- | --- |
| Approach | Benefits |
| **Document workload features** and capture customer benefits.  Derive **scope and detailed functional and nonfunctional requirements** of the architecture.  **Create sizing estimation models** to report on scope and cost of the tasks involved. | Good specifications **cut operational costs and chances of failure** by supporting more productive and streamlined development cycles.  Developers understand the **technical design, goals, and completion criteria** before they start the coding cycle.  Good documentation facilitates repeatable **communication** and **onboarding** of new team members. |
| Use an **industry standard software development methodology that's appropriately tuned** for the needs of your workload and team size.  **Maintain a backlog that's shared among all roles**. | Adoption of a well-known methodology **sets the rhythm of the project**. It removes process ambiguities by giving team members clear expectations and accountability.  By tracking against a common list, **tasks can be refined and prioritised** with standard practices. The project will have better chances of being delivered on time.  Standard methodologies help with **risk management**. With granular milestone reviews, developers can address potential issues before they become showstoppers. |
| **Use unified source control** for all code, scripts, deployment templates, pipeline definitions, and related documentation.  The branching strategy must support friction-free release of independent and interdependent features, bug fixes, and hotfixes.  Use shared knowledge across the organisation to build your branching strategy and deployment processes. | Proper use of source control is crucial in supporting **concurrent changes** and versioning.  Maintain a repeatable workflow for releasing changes of various sizes and risks, conduct peer reviews as part of the process, and keep an audit trail. |
| Have **quality assurance** processes that emphasise testing early in the development lifecycle.  **Include all artifacts for planned test procedures**, including application components, infrastructure, and data plane operations that are part of a feature release or update.  Treat artifacts as immutable when they're promoted through environments, gaining confidence each time they pass through a quality gate.  Where practical, automate routine checks. | Quality assurance ensures that functional and nonfunctional requirements were met with confidence, which leads to positive customer impact.  Having test plans ensures quality and completeness and takes possible failure cases into consideration.  With quality gates, you can enforce best practices to reduce risks.  Immutability brings confidence because it ensures the system that you test is exactly what you release.  Testing cycles efficiently block progress unless quality criteria are met. |
| Drive consistency by using **style guides and tools**, which enforce **conventions**, and adopt a **common tool chain** for development, testing, and communication with stakeholders.  **Technology standards for developers should** necessitate implementation **of patterns**, API design, **logging, exception handling, and other processes**. | Consistency in code drives readability and easier maintenance. It also reduces complexity and enables code reuse.  Common tooling and conventions also help teams optimise processes without the need to address one-off choices. |
| Consistently and deliberately insist on developer documentation of code as its written. | Clear code documentation ensures that logic and functionality are easily understood when old code needs to be revisited or when development teams rotate. |
| **Report progress and trends** to measure efficiency. | Trends in bugs, failed updates, time to deploy, feedback loops, and other metrics are published, and that drives improvements. |

**Improve Operations with Monitoring and Insights**

**Gain visibility into the workload and use insights to make data-driven decisions.**

Continuously improve the quality of your workload by monitoring it in accordance with Power Platform Well-Architected pillars. Build a monitoring system that tracks every aspect of the workload. Collect data that you can learn from and that can drive improvements. Monitoring is key in proactive maintenance of the workload, quality and security assurance, performance and capacity planning, and product management. Efficient monitoring can reduce reactive cycles and incident response times.

|  |  |
| --- | --- |
| Approach | Benefits |
| **Build a monitoring system with its own stack and flows**.  Treat the monitoring system as a dimension of the workload that's decoupled from its utility. The stack must cover all layers, including infrastructure, application health, and build and release processes.  **Capturing or sampling business data** is out of scope for observability implementations. | Decouple monitoring and workload stacks to **separate functional requirements and observability requirements** and make independent evolution possible. Changes in code shouldn't affect monitoring, and vice versa.  Because observability requirements are separate from functional requirements, **business data** **won't be disrupted** by monitoring configuration changes or outages. |
| **Drive consistency** in the collection process for each **type of data source**.  Standardise instrumentation in code by using industry standards for telemetry, collection of infrastructure metrics, and tooling. | Consistency prevents variance in sensing and measurement because familiarity across similar resources **reduces time spent correlating and analysing data**. You have a holistic perspective to anticipate issues. |
| **Emit telemetry** from application code that correlates the key points of the execution flow and gives an end-to-end view at different levels of granularity. | Prioritise actions based on the severity level, and understand the context given its verbosity. This information is crucial for troubleshooting purposes. |
| **Own the responsibility of emitting and collecting data**, even when data sinks are shared by multiple teams and managed by central teams. | By localising monitoring data to the workload environment, the team can access logs and metrics to address workload concerns. |
| Collect **just enough data** and retain it for **just enough time**.  Consider the cost tradeoffs associated with logging and storing data. | Intentional data collection helps you **optimise financial and operational costs** associated with collecting more data than you need.  **Minimise the noise and avoid intensive computation** during analysis, and reduce the cost of storing data that you no longer need. |
| **Make a distinction between the different monitoring signals**: profiles, logs, metrics, and traces. Use each signal for the right purpose.  Prioritise the **use of metrics to trigger actions** that rely on numeric measurements.  Use **profiles to get lower-level visibility**, such as memory allocation, into the system.  Reserve the **use of logs and traces to provide context** for flows and dependencies. | By using the signals for the right purposes, you can prevent inefficient implementation of the monitoring system.  For example, using logs for actions requires parsing. You might be able to achieve the same goals faster with metrics. |
| **Aggregate and visualise data** in dashboards to present monitoring data that's catered to audiences and keeps the business context in mind.  Use **situational dashboards** for surfacing data to drive awareness among the stakeholders.  **Use operational dashboards and workbooks** with drill-down capabilities for operator activities like incident response. Frequently refresh the dashboards and provide granular data. | With visualisations, you can Analyse trends, track against business targets, and manage incidents.  Dashboards that are tailored to the interest of the customer make interpretation relevant and accelerate time to detection and action. |
| **Make alerts actionable** by notifying the accountable roles with Standardised descriptions and severity levels. Provide information that's collated from various sources and track deviations from business targets.  **Trigger alerts only for incidents that require action**.  **Strive for proactive** and thought-provoking alerts that initiate actions before a degraded state becomes a failure. | Alerts bring attention to significant events as defined by the organisation.  A good alert system identifies actions and severity and provides just enough data to drive clarity and purpose. **Operators can start on remediation without delay**. |

**Deploy With Confidence**

**Reach the desired state of deployment with predictability.**

Build a workload supply chain that enables you to consistently deploy your workload to all your environments. Choose tooling that is capable of automation, testing, monitoring, and versioning. Upskill your team so that they can use the chosen technology and increase productivity. Aim to achieve immutable assets that deploy through automation and avoid configuration drift and manual changes in downstream environments.

|  |  |
| --- | --- |
| Approach | Benefits |
| **Use Infrastructure as Code (IaC)** to define the repeatable aspects of the supply chain that are production ready.  Prefer declarative approaches over imperative methods. | Declarative IaC technologies are designed with automation and reusability in mind. You can offload infrastructure deployments from individuals into tooling and achieve consistent quality.  From an infrastructure perspective, having fewer technology choices removes variance in tooling and makes configuration drift easy to detect. Maintenance will also be easier. If you align choices with the team's existing skill set, the team can easily adopt them. |
| **Prepare the team to use the chosen IaC technology**. Learn about its extensibility model, capabilities, and limitations.  Take advantage of specialisation within the team and shared knowledge within the organisation. | Upskilling increases productivity and fosters an environment of collaboration through shared learning.  You can fill gaps with training instead of hiring. |
| **Follow software recommendations** for IaC development and maintenance.  Modularise in moderation. Avoid custom or low-value abstractions.  Follow a layered approach to reflect different lifecycles. Form foundational layers where the lower layers stay constant and the upper layers change as needed.  Deployment artifacts, such as application binaries, IaC templates, and parameters, are part of the attack surface. Apply assurances, such as secret management, access control, and other principles of the Security pillar. | Artifacts experience the same level of engineering rigor as application code. Quality controls through peer reviews and testing give you confidence in deployment.  A layered approach makes maintenance easier and creates boundaries that establish clear lines of responsibility.  Adding security controls to artifacts helps harden the system during the deployment process. |
| Develop a **common deployment** manifest that's used across all environments. Use that manifest as the default mechanism for greenfield projects, incremental workload updates, or disaster recovery. | Remove the overhead of maintaining multiple assets.  If there's a disaster, recovery will be quick and reliable because you can deploy a tried and tested manifest instead of creating an improvised environment. |
| Strive for **immutable and ephemeral infrastructure** that's deployed through IaC automation. | Prohibit configuration drift and make the deployment idempotent.  This kind of infrastructure removes significant operational burdens, such as patching. It also benefits core validation scenarios, such as blue-green infrastructure deployments. |

**Automate For Efficiency**

**Replace repetitive manual tasks with automation designed to complete tasks quicker, with greater consistency and accuracy, and reduce risk.**

Evaluate repetitive tasks that might be time-consuming and error prone. Identify opportunities for automation to free up your team for higher-value tasks, increasing productivity and consistency. Design your automation with the same principles that you apply to your overall workload, avoiding anti-patterns and keeping security threats in mind.

|  |  |
| --- | --- |
| Approach | Benefits |
| **Evaluate all workflows** against criteria that's at the right level of complexity, effort, frequency, accuracy, timeliness, and lifespan.  **Automate workflows** based on that evaluation and prioritise the workflows with the highest expected returns.  **Remove redundant workflows** or add value to justify human effort. | You can reinvest team capacity in higher value work and increase productivity and consistency.  Building an inventory of workflows ensures that you automate the right tasks. Removing redundant tasks reduces complexity and errors. |
| Be explicit about your decision when you evaluate whether to **build custom tooling or buy software**.  Reserve building automation for highly specialised and high-value work. | By buying off-the-shelf software and taking advantage of the support contract, you save on maintenance costs.  By building software, you have more control and can cater to use cases that are unique to your team and workload. However, there's a cost impact.  Choice of tooling brings a level of Standardisation to your operations. With training, you can achieve a uniform level of readiness for adoption. |
| Design your workload components to **support automation capabilities**. | Avoid the situation where lack of automation in your system design promotes the anti-pattern of repetitive tasks, slows down growth, and starts accumulating technical debt. |
| Treat all **automation as a critical dependency** of your workload. Adapt to the workload's expected growth.  Your automation tooling is an integral part of your workload, and it should adhere to the five Well-Architected Framework pillars. | Design your automation component to withstand risks, such as security threats. With applied best practices, you can avoid implementation sprawl.  The workload will continue to operate with a high-level guarantee if this dependency is kept functional and safe. |
| **Automate at-scale** by exploring options beyond your workload.  **Favor a "design once, run everywhere"** model by providing templates and frameworks to onboard new projects and promote reuse of existing designs and implementations. | Employ tried and tested methods and reduce chances of failure. |

**Adopt Safe Deployment Practices**

**Implement guardrails in the deployment process to minimise the effect of errors or unexpected conditions.**

Standardise deployment of any changes by using automated deployment processes, such as pipelines. Consistent deployment methods ensure the deployment is completed safely, reliably, and with repeatability. Catch issues in early stages of deployment by testing changes rigorously. Ensure you have a mitigation strategy in place to quickly recover from deployment failures.

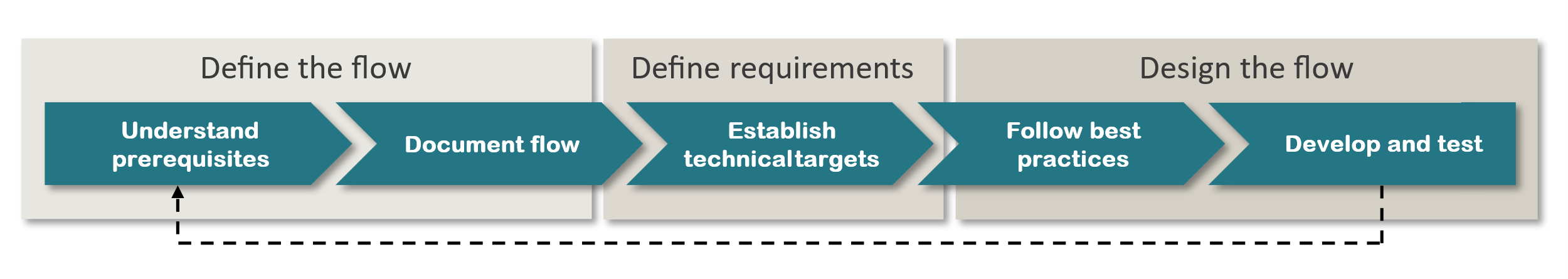
|  |  |
| --- | --- |
| Approach | Benefits |
| **Standardise the process to deploy any change by using automated deployment processes, such as pipelines**.  All environments must use pipelines.  Classify assets and versions per environment to make them **easily** traceable and identifiable. | **Consistent deployment methods reduce** issues caused by process errors and variance and allow you to focus your effort on the workload concerns.  Standardisation ensures that the deployment is completed safely, reliably, and with repeatability.  Classification makes it easy to view logs of previous deployments and issues that have occurred. You might be able to use that information to expedite rollback and roll-forward operations. |
| Deploy **small incremental updates** at a regular cadence. | Frequent, well-tested, small updates make **validation of the release easier**.  Troubleshoot faster with **minimal customer impact** due to a smaller footprint. |
| **Test updates rigorously by using different mechanisms** throughout the development lifecycle. | **Catch issues in the early** stages of development. Iterative fixes and consistent deployment practices cause issues to taper off by the time the update is ready for production. |
| **Roll out updates gradually, with due diligence**.  Use deployment models that give you the control to **progressively increase the number of instances and customers** until the update is safely adopted by all. | Test each update in a controlled manner so that issues are fixed early in production. Avoid rolling out a faulty update that impacts your entire customer base.  Test whether the update is backward and forward compatible. |
| Have a mitigation strategy to quickly **recover from deployment failures**.  The strategy should cover decision making on **rolling back or forward** based on the criticality of the issue.  Have **well-defined processes and automated systems** that can rapidly roll out fixes by using the standard deployment pipelines. | Reduce the duration of potential impact.  Restore the system back to the previous working version or roll forward to a version that has fixes that have been thoroughly tested. |
| Have a fallback plan **that resets the system** to a working state in case of emergency and to recover from unexpected failures. Use this strategy only when necessary and with approval.  Strive to improve the plan over time. | You can fast-track high-priority fixes, such as security remediation.  The accelerated pipeline might not have all the checks of your standard operating procedures, but you'll get customers to a safe version in the fastest way possible, which outweighs lower-impact faults. |

## Flow Optimisation

This section covers the targeted optimisation of workloads using flows. Different components of a workload have varying requirements and levels of importance. By segmenting a workload into flows, you can prioritise different parts of a workload and better align workload investments with the importance of each flow.

This workload optimisation process is iterative and involves three key steps: (1) define the flow structure within your workload, (2) define technical requirements, and (3) design the flow to meet the requirements (*see figure 1*).

*Figure 1: The process to optimise workloads using flows.*



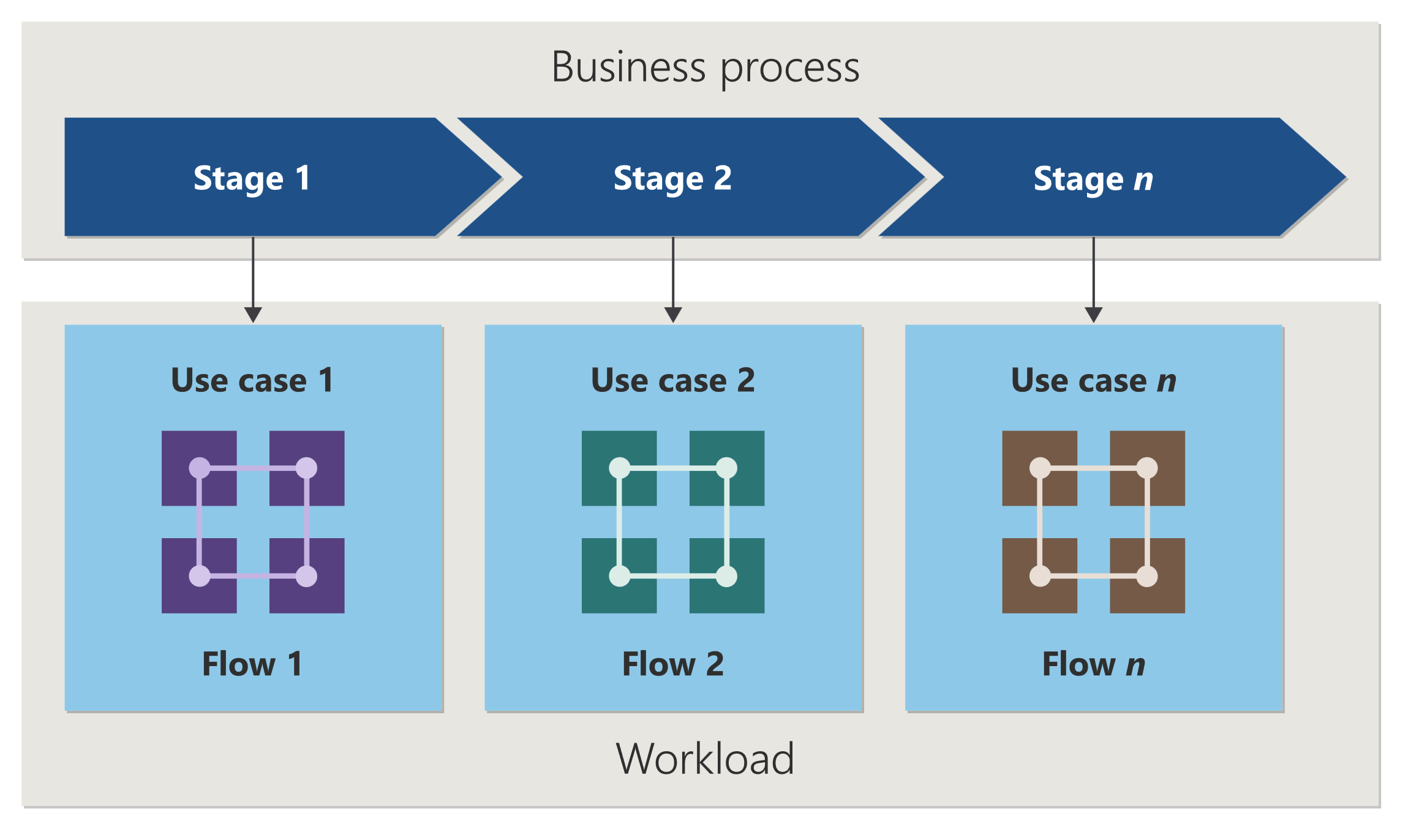
**Define the flow**

Before you can define flow requirements, you need to understand the business drivers for the flow. The prerequisites to defining a flow are identifying the business process and use case its supports. When you understand the prerequisites, you can start documenting the flow.

**Understand the prerequisites**

Flows are sequences of actions that support workload functionality. There are two primary types of flows: user flows and system flows. User flows determine user interactions. System flows determine communication between workload components. Flows support business processes and use cases. A workload consists of multiple use cases. You need to identify the business process and use case the flow supports before documenting a flow (*see figure 2*).

*Figure 2: The relationship between business processes, use cases, flows, and workload.*



**Identify the business process**

A business process is a series of actions (stages) that fulfill a business requirement. Flows determine the sequence a user or data takes to accomplish each stage of a business process. For example, selling products online is a business process. The stages in this business process might be listing the product online, receiving orders, and delivering the product.

**Identify the use case**

A use case defines the functional requirements of a flow. You need to identify and understand the use case a flow supports before establishing the technical requirements of a flow. Each use case should support one stage in a business process (*see figure 2*). A use case should define the following attributes:

* ***Purpose***: Clearly articulate the tasks or objectives, like enabling online purchases. This clarity guides the functional design and sets clear goals for flow design.
* ***Criticality***: Assess the importance of the use case, ranging from routine to critical. The value assigned to a use case informs the prioritisation and design of the flow. High-value use cases might require enhanced error handling, performance tuning, or user experience considerations.
* ***Consumers***: Identify whether users (customers, staff) or system components are the primary consumers. This categorisation determines whether it's a user flow or system flow and influences the design.
* ***Events***: Define triggers or conditions that initiate and conclude the use case. These events define the flow's boundaries.
* ***Execution***: Understand the operational frequency and variability of the use case to anticipate system load. You must design a flow to handle different execution scenarios.
* ***Dependencies***: Identify interdependencies with other use cases for risk management. Recognising a use case's dependencies aids in designing flows that integrate smoothly with other system parts. You need to ensure the availability of necessary inputs and compatibility of outputs with subsequent processes.

**Document the flow**

Use the use case to document the flow. You should outline or map each action you need in a flow. Capture decision criteria and pathways. Identify interactions with other use cases. This outline serves as a blueprint for flow design and management. You also need to capture business information about the flow. Make sure to include the following details in the flow documentation:

* ***Flow description***: A high-level description of the flow.
* ***Business******process***: The business process that the flow supports.
* ***Process owner***: The individual that owns the business process.
* ***Stakeholders***: The individuals that you should inform or consult on flow status or changes.
* ***Escalation paths***: The individuals or groups you should contact to resolve issues. It's a sequence of people. The scope of individual responsibility grows with each person on the path.
* ***Business impact***: The importance of this flow to the business.
* ***Criticality rating***: A qualitative label that indicates the relative importance of the flow.

**Define flow requirements**

Utilise the use case to establish the technical targets of the flow. Define measurable targets for the flow that align to the five pillars of the Well-Architected Framework (WAF). These pillars provide a framework for setting technical targets:

* ***Reliability targets***: Assess each flow's importance and set reliability targets accordingly. Determine performance thresholds and establish clear service level agreements (SLAs) and objectives (SLOs). Higher criticality flows require more stringent reliability targets.
* ***Security targets***: Analyse the security needs of each flow based on data sensitivity and user activities. Implement and continuously update security measures to meet these needs while ensuring compliance with regulatory standards.
* ***Cost targets***: Understand the demands of each flow for effective resource allocation. Set targets to balance cost with performance. Ensure resource usage aligns with business priorities.
* ***Operational targets***: Define metrics for effective monitoring and troubleshooting. Targets should ensure efficient resource use and alignment with organisational goals.
* ***Performance targets***: Base performance targets on the initial requirements of each flow. Ensure that essential flows receive adequate resources and continuously adjust targets to meet evolving demands and enhance user experiences.

**Design the flow**

Design the flow to meet the technical targets. You should familiarise yourself with flow design best practices so that you achieve the right result. Build and test the flow. Iterate on the design until it meets the technical targets you established.

**Follow flow design best practices**

As you design a flow, follow flow design best practices. A well-designed flow has the following attributes:

* ***Scoped***: Identify distinct starting and ending points for each flow. Clear boundaries help optimise user or system interactions.
* ***Logical****:* Design your flows with a logical order of steps. Optimise for the most efficient path and reduce unnecessary steps.
* ***Maintainable***: Design flows that are easy to update and maintain. Use modular components that you can modify without affecting the entire workload.
* ***Defined***: Incorporate specific conditions that trigger or guide each step in a flow. This precision ensures that the flow responds accurately to user inputs, data changes, or system states.
* ***Reliable***: Build error handling and exception paths into your flows. Effective error management prevents disruption and maintains flow integrity under unexpected circumstances.
* ***Scalable***: Ensure it can handle varying loads and adapt to growing or shrinking user bases or data volumes.
* ***Secure***: Embed security measures within the flow. Protect data and user interactions against unauthorised access and threats.
* ***Efficient***: Plan for efficient use of resources without over-provisioning. Keep cost optimisation in mind.
* ***User-centric***: For user flows, align the flow design with user expectations and behaviors. Make it intuitive and reduce the learning curve for new users.

**Develop and test the flow**

Develop the flow to meet technical targets and test it to ensure it meets its requirements. This process validates that the flow operates as intended, efficiently handles its tasks, and meets the technical targets. Here's guidance to build and test a flow:

* ***Select technologies***: Choose technologies that align with the set targets in terms of reliability, security, and performance.
* ***Develop flow***: Build the flow according to the design, keeping the set targets in mind.
* ***Test flow***: Conduct testing to ensure the flow meets targets. iterate as needed to meet targets.
* ***Monitor***: Implement monitoring tools to track resource usage and costs.

Periodically review the flow against set targets and industry standards. Use feedback from monitoring and audits to improve the flow. Adjust targets and processes as necessary to align with changing business needs or technological advancements.

**Optimise flows**

Repeat the process defined in this section throughout the lifecycle of the flow. As you iterate on the flow design, use the Well-Architected Framework to optimise flows from the perspective of each pillar:

* [Flow reliability](https://learn.microsoft.com/en-us/azure/well-architected/reliability/identify-flows)
* [Flow security](https://learn.microsoft.com/en-us/azure/well-architected/security/secure-development-lifecycle#requirements-phase)
* [Flow cost optimisation](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-flow-costs)
* [Flow operational excellence](https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/observability#key-design-strategies)
* [Flow performance efficiency](https://learn.microsoft.com/en-us/azure/well-architected/performance-efficiency/prioritize-critical-flows)

**System flow: Collect telemetry**

*Flow description*: To understand state changes in the production system, web application and API instances collect and send information, errors, and warnings. This data helps the operations team perform anomaly detection, troubleshooting, and profiling.

* ***Business processes***: This flow doesn't support any business processes, but it provides important data for the operations team.
* ***Process owner***: Director of Operations.
* ***Stakeholders***: Operations team, platform team, and data team.
* ***Escalation path***: Operations team (24/7), data team on-call engineer.
* ***Business impact***: This flow is essential for the business's monitoring and continuous improvement efforts. It needs to be as redundant and resilient as possible. The operations team is responsible for quickly restoring this flow after any failure to avoid missing critical information and warnings. If the flow fails to achieve the expected availability, there's a risk of overlooking production issues, potentially leading to severe consequences. To mitigate this risk, the operations department aims for 99% uptime, 24/7. They must schedule maintenance-related downtime at least 48 hours in advance.
* ***Criticality rating***: Medium.

## Checklist

* Develop robust deployment processes.
* Implement continuous monitoring.
* Maintain detailed documentation.
* Conduct regular process reviews.

This checklist presents a set of recommendations to help you build a culture of operational excellence. Start with a fusion development and DevOps approach to integrate specialisations from multiple disciplines. This approach creates a rigorous design and development practice that leads to repeatable, reliable, and safe deployments of infrastructure and code.

Prioritise human intervention in areas that benefit from it and incorporate automation in other areas. Observability serves operational excellence by monitoring health events and validating the current workload design and implementation to inform future product development.

If you don't consider trade-offs and recommendations for operational excellence, your workload might be at risk. Carefully consider the points covered in the following checklist to instil confidence in your design's success.

|  |  |
| --- | --- |
| Code | Recommendation |
| [OE:01](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/fusion-culture) | **Determine workload team members' specialisations and integrate them into a robust set of practices** to design, develop, deploy, and operate your workload to specification. Team members must have clarity in decision making and responsibilities, value continuous improvement and optimisation, and adopt a blameless culture that incorporates continuous learning. |
| [OE:02](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/formalize-operations-tasks) | **Formalise the way you run routine, as needed, and emergency operational tasks** by using documentation, checklists, or automation. Strive for consistency and predictability for team processes and deliverables by adopting industry-leading practices and approaches, such as a "shift left" approach. |
| [OE:03](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/formalize-development-practices) | **Formalise software ideation and planning processes.** Draw from established industry and organisational standards. Use a common, prioritised backlog and sufficiently detailed specifications. Based on outcomes, drive continuous improvements in your planning process. |
| [OE:04](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/tools-processes) | **Optimise software development and quality assurance processes** by following industry-proven practices for development and testing. For unambiguous role designation, Standardise practices across components such as tooling, source control, application design patterns, documentation, and style guides. |
| [OE:05](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/workload-supply-chain) | **Build a workload supply chain that drives proposed changes** through predictable, automated pipelines. The pipelines test and promote those changes across environments. Optimise a supply chain to make your workload reliable, secure, cost-effective, and performant. |
| [OE:06](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/observability) | **Design and implement a monitoring system** to validate design choices and inform future design and business decisions. This system captures and exposes operational telemetry, metrics, and logs that are emitted from the workload. |
| [OE:07](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/emergency-response) | **Develop an effective emergency operations practice.** Ensure that your workload emits meaningful health signals. Collect the resulting data and use it to generate actionable alerts that enact emergency responses through dashboards and queries. Clearly define human responsibilities, such as on-call rotations, incident management, emergency resource access, and running postmortems. |
| [OE:08](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/automate-tasks) | **Automate all tasks that don't benefit from the insight and adaptability of human intervention, are highly procedural, and have a shelf-life that yields a return on automation investment.** When possible, choose off-the-shelf software for automation versus custom implementations. Treat all automation the same as workload components and apply the Power Platform Well-Architected pillars to its design and implementation. |
| [OE:09](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/enable-automation) | **Design and implement automation up front** for operations such as life cycle concerns and apply governance and compliance guardrails. Don't try to retrofit automation later. Choose automation features that your platform provides. |
| [OE:10](https://learn.microsoft.com/en-us/power-platform/well-architected/operational-excellence/safe-deployments) | **Clearly define your workload's safe deployment practices.** Emphasise the ideals of small, incremental, quality-gated release methods. Use modern deployment patterns to control risk. Account for routine deployments and emergency, or hotfix, deployments. |
| OE:11 | **Implement a deployment failure mitigation strategy** that addresses unexpected mid-rollout issues with rapid recovery. Combine multiple approaches, such as rollback, feature disablement, or using your deployment pattern's native capabilities. |

## Trade-offs

Operational Excellence supports workload quality through the implementation of clear team standards, understood responsibility and accountability, attention to customer outcomes, and team cohesion. The implementation of these goals is rooted in DevOps, which recommends minimising process variance, reducing human error, and ultimately increasing the return of value for the workload. That value isn't only measured against the functional requirements served by the components of the workload. It's also measured by the value that the team delivers in striving for improvement.

During the design phase of a workload and over its life cycle as continuous improvement steps are taken, it's important to consider how decisions based on the Operational Excellence design principles and the recommendations in the design review checklist for Operational Excellence might influence the goals and optimisations of other pillars. Certain decisions might benefit some pillars but constitute trade-offs for others. This section describes example trade-offs that a workload team might encounter when designing workload architecture and operations.

**Operational Excellence trade-offs with Reliability**

Reliability prioritises simplicity, because simple design minimises misconfiguration and reduces unexpected interactions.

* Safe deployment strategies often require some amount of forward and backward compatibility between application logic and data in the workload. This added complexity increases the testing burden and can lead to complexities or integrity issues with the workload's data.
* Highly layered, modularised, or parameterised structures can increase the chance of accidental misconfiguration due to the complexity of the interaction between the components of the workload.
* Cloud design patterns that benefit operations sometimes require the introduction of more components, for example, the use of secret store or taking a dependency on Application Insights. The additional components increase the points of interaction in the system, increasing the potential for malfunction or misconfiguration.

The Reliability pillar encourages the avoidance of activities or design choices that can destabilise a system and lead to disruptions, outages, or malfunctions.

* Deploying small, incremental changes is a technique for mitigating risk, but users expect those small changes to be delivered to production more frequently. Deployments can destabilise a system, so as the rate of deployment increases, so does this risk.
* A culture that measures itself with velocity metrics like deployments per week and uses automation that can facilitate introducing changes at a faster pace is also likely to perform more deployments in a shorter period.
* Increasing density to simplify operations by reducing the number of control and observability points can also lead to an increased availability risk because malfunction or misconfiguration increases the effect of a destabilising event.

**Operational Excellence trade-offs with Security**

The Security pillar recommends a reduced workload surface area in terms of components and exposure to operations. This reduction minimises vulnerabilities and produces a smaller scope for security control and testing.

* Components that surround the workload and support its operations, like automation or a custom control plane, must also be in scope for regular security hardening and testing.
* Routine, unplanned, and emergency operations increase the points of contact with the workload. A zero-trust approach requires that these processes are considered vulnerabilities and must be included in the security controls and validation for the workload.
* The observability platform of the system collects logs and metrics about the workload, which can be a valuable source of information disclosure. Therefore, the workload's security needs to extend to protect data sinks from internal and external threats.
* Build agents, externalised configuration, and feature toggle stores all increase the application surface area that requires security.
* A higher deployment frequency caused by small, incremental changes or by "get current, stay current" efforts results in more security testing in the software development life cycle (SDLC).

**A secure workload is based on designs that protect the confidentiality of data that flows through the components of the system.**

Observability platforms ingest data of all types to gain insights into a workload's health and behaviour. As teams try to attain higher fidelity in observability data, there's an increased risk that data classification controls, like data masking, of the source systems don't extend to the logs and log sinks of the observability platform.

**A key security approach for isolating access and function is to design a strong segmentation strategy. This design is implemented through resource isolation and identity controls.**

* Co-locating disparate application components in shared environments and data resources to make management easier reverses segmentation or makes role-based segmentation harder to achieve. Co-located components might also need to share a workload identity, which can lead to over-assignment of permissions or a lack of traceability.
* Collecting all logs from across the system in a unified log sink can make querying and building alerts easier. However, doing so can also make it harder or impossible to provide row-based security to treat sensitive data with the required audit controls.
* Simplifying the management of attribute-based or role-based security by reducing the granularity of roles and their assignments can lead to inappropriately broad permissions.

**Operational Excellence trade-offs with Experience Optimisation**

**The Experience Optimisation pillar recommends a user-centric mindset.**

* Users experience development that requires significant resources might be deprioritised, which can cause the experience to lack the usability, interactions, and visual design that workload users need.
* User interface development is often done in faster iterations and ship cycles, which can strain the team's SDLC processes.

**Operational Excellence trade-offs with Performance Efficiency**

**The Performance Efficiency pillar recommends allocating as much of the available compute and network resources as possible to the requirements of the workload.**

* A workload's monitoring framework requires that the components in the architecture allocate time and resources to create, collect, and stream logs and metrics. These data points help ensure that effective alerting and monitoring is possible for reliability, security, and performance. As the level of instrumentation increases, the strain on system resources might also increase.

To create performant workloads, teams look for ways to reduce the time and resources that workloads consume to perform their tasks.

* Some cloud design patterns that support "independent change over time" approaches to support the ideals of incremental improvement can introduce latency due to the traversal of additional components.

## Develop Robust Deployment Processes

Establish standardised deployment pipelines using tools like Azure DevOps or GitHub Actions. Ensure deployments are repeatable and consistent across all environments.

## Implement Continuous Monitoring

Set up monitoring tools to track system performance and health in real-time. Use automated alerts to notify teams of any issues promptly.

## Maintain Detailed Documentation

Document all operational processes, including deployment, monitoring, and maintenance procedures. Ensure documentation is accessible and regularly updated.

## Conduct Regular Process Reviews

Schedule periodic reviews of all operational processes to identify areas for improvement. Use metrics and feedback from stakeholders to drive continuous improvement efforts.

# Performance Efficiency

Performance is a key factor in the success of any Power Platform workload. Poor performance can frustrate users, reduce productivity, and increase costs. To avoid these problems, you need to design your solutions with performance in mind from the start. A performant workload is able to handle changes in load without compromising the user experience or exceeding throughput and request limits of the platform.

Use the following design principles from the [Microsoft Azure Well-Architected Framework](https://learn.microsoft.com/en-us/azure/well-architected) to help build a strategy that ensures the performance meets your business requirements whilst at the same time staying within the limits and capabilities of Power Platform services. Start with the recommended best practices and justify the benefits of each performance efficiency principle. Once you set your strategy, use the [Performance Efficiency checklist](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/checklist) as your next step.

Without a clear understanding of the performance expectations of your workload, you might end up spending too much or too little on resources and fail to satisfy user needs.

## Design Principles

* **Scalability**: Design systems to scale efficiently in response to varying workloads.
* **Optimisation**: Continuously optimise system performance through tuning and enhancements.
* **Resource** **Management**: Manage resources effectively to balance performance and cost.

**Negotiate realistic performance targets**

**The intended user experience is defined, and there's a strategy to develop a benchmark and measure targets against the pre-established business requirements.**

Start your design process with clear performance targets based on the business needs and expectations. Rather than just focusing on technical metrics, work with business stakeholders to set expectations and define targets that are aligned with the expected user experience of the workload.

|  |  |
| --- | --- |
| Approach | Benefits |
| **Prepare for effective negotiation** by understanding technical concepts, exploring design possibilities with the available infrastructure, and using results from concrete experimentation, if available.  **Use historical data** to get visibility into usage patterns and bottlenecks.  Bring insight from external factors, such as **input from market analysis, experts, and industry standards**. | You can make **informed decisions** based on practical insights.  The performance targets are focused on user experience that's based on what's feasible, industry best practices, and current market trends. |
| **Collaborate with the business owners** to understand user promises, in terms of quality and regulatory compliance, if applicable.  **Maintain a broad perspective** and avoid diving into granular details at this stage.  Be explicit about **what represents acceptable performance**, based on the investments.  Understand the business context and **anticipated growth**. | You'll **avoid making assumptions** that might not align with the business goals. It also drives clarity and motivation within the workload team.  Having a business context on functional and nonfunctional requirements might uncover design changes in other Azure Well-Architected pillars and **help you make informed tradeoffs**.  Defining parameters early on helps avoid costs associated with potential solution redesigns later.  It enables you to ensure that **performance targets cover future projections**, so you can align current efforts with long-term goals. |
| **Identify the workload flows** and prioritise the flows in the architectural diagram.  **Define each flow's performance tolerance** as a range from aspirational to unacceptable performance.  **Evaluate the entry and exit points for each flow**, considering the path's criticality, usage frequency, and architectural intensity. | By prioritising flows, you can **focus your resources on critical areas** that have the most effect on user and business outcomes.  By breaking down the system into its parts and dependencies, you understand each component's function and influence on performance. You also become aware of potential issues.  It helps establish a performance baseline and drive optimisation. |
| **Start building a performance model** Consider whether usage patterns show seasonal or daily variations. Factor in the cost, operations, and criticality to the business.  **Use industry standards to quantify metrics** and aggregation methods, such as using percentiles.  **Evaluate the demand and supply expectations** and limitations that business constraints impose.  Incorporate growth prospects. | A performance model provides **insight into optimal use of resources** and helps with strategic planning.  Industry standards help with benchmarking.  Future proofing ensures that the performance goals remain relevant and can adapt to changes. |

**Design to meet performance requirements**

**Select the right services to meet performance targets.**

It's important to proactively measure performance. Choose services and features across the technology stack that enable you to meet your performance goals, monitor performance, and track which components of the workload might pose challenges. Also, define a process for testing performance.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Evaluate the elasticity demands** for the identified flows.  **Explore design patterns** that can be implemented across the technology stack, considering the application and the underlying compute and data layers. | You're able to **define scalability requirements** on existing components that need more capacity and the areas where you need extra components to distribute load.  You're aware of potential bottlenecks in the system and **design compensating controls**, such as adding caching capabilities to decrease latency and system load. |
| **Choose the right resources** across the technology stack, which enables you to meet performance goals and integrate with the system.  **Consider features** that can fulfil the scalability requirements.  **Find the right balance between resource allocation and system requirements**, to manage unexpected surges efficiently. | By analysing the varying capabilities of the resources, you ensure that each component contributes to the **overall functionality and performance of the system**.  You can **take advantage of the built-in capabilities** that automatically trigger scaling operations.  Right-sizing resources can meet changes in demand without overprovisioning, which leads to cost savings. |
| **Do capacity planning** based on demand and the capability of selected resources to enrich your performance model.  **Use predictive modelling techniques** to forecast anticipated changes in capacity that can occur with predictable and unexpected changes.  **Define performance targets** that can be translated into technical requirements. | You can **efficiently use resources** and meet the demand without overprovisioning, thereby avoiding unnecessary costs.  You understand how the design choices affect performance. |
| **Implement a proof of concept** that validates the technical requirements and design choices. | A proof of concept is instrumental in **validating the design** to determine if the system can meet the performance targets and if those targets are realistic. Based on the anticipated load, you can validate whether anticipated capacity can meet the performance targets.  Also, verify the cost implications of the design choices. |
| **Document your performance testing strategy**.  Include use cases, different methodologies, and cadence of your test plans.  Define a process for operation outlined by the performance test plan.  Triage and prioritise the test cases in the plan. Focus on cases that offer valuable insights into performance targets and align capacity planning. | You ensure that the **right aspects of the system are tested**.  You can allocate resources effectively and conduct tests in a manner that aligns with the business priorities and requirements. |
| **Document your performance monitoring strategy**.  Assess metrics at different abstraction levels for each identified flow. | You can **track progress towards attainment** of performance targets throughout the development cycle. |

**Achieve and sustain performance**

**Protect against performance degradation while the system is in use and as it evolves.**

Performance isn't a one-time activity. You need to keep working on it throughout the development of the workload. Expect to test and optimise your workload multiple times. Any changes to requirements, configuration, code, or product features can affect performance.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Integrate routine performance tests** in Azure Pipelines.  Choose pipelines that can integrate tests. Conversely, choose test tools that can be integrated into the pipelines. | Automated tests save time and provide consistency that makes it **easier to detect regressions or improvements**.  These artifacts allow for continuous monitoring of any deviations or drift over time, so you can maintain consistent performance and quality. |
| **Formalise performance tests as quality gates** that can approve or deny release promotion and the final deployment to production. | These checkpoints ensure that **each stage of deployment meets the required performance standards** before you proceed to the next. The checkpoints help prevent unintended performance regression.  For instance, if the performance is significantly below expectations, you might block a release until improvements are made. |
| **Set up a repeatable process for monitoring** real transactions in production and deviations against your performance targets.  Use synthetic transactions in production.  Set up monitoring alerts on performance regressions. | You want insight into the **actual performance of your system under real-world load** that could not be simulated through tests.  Then you can proactively identify issues and areas of improvement such as potential bottlenecks, underutilised resources, and other concerns. |
| **Review performance test results and monitoring data** meticulously and optimise until you meet the performance targets.  Prioritise actions derived from those reviews and add them to the backlog for planned execution. | Based on test results, **you can capture and compare data** and start analysing trends.  Your optimisation efforts are data driven. |
| **Build coding skills that focus on performance**.  Have coding standards that exemplify performance-driven coding patterns. | Code that does not have performance issues can make **testing cycles more efficient** because tests can be focused on more significant issues.  Coding patterns helps avoid rework and keeps your coding style consistent. |
| **Address performance erosion** as usage increases, features change, and data accumulates over time to sustain performance.  Reset expectations and establish new targets, if fine-tuning brings only short-term benefits. | You can **preserve the performance state** before degradation develops into problems that negatively affect user experience beyond the acceptable range.  Changing targets resets the performance model, and you don't waste time in optimising the system that has already reached its capacity. |

**Improve efficiency through optimisation**

**Improve system efficiency within the defined performance targets to increase workload value.**

Establish a performance culture that allows developers to spend time on performance optimisation. Adjust performance targets based on user experience, and monitor which components of your workload experience heavy load. Continue to evaluate new product features that could improve performance. The cycle of monitoring, Optimising, testing, and deploying is a continuous process.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Allocate dedicated cycles for performance optimisation** to address nonfunctional requirements and optimisations in functional areas. Targets for this optimisation are resources, code, data retention, database queries, and others. | You can **build a culture of performance-driven optimisation**. You keep the team accountable for proactively monitoring performance patterns and also fine-tune the application. |
| **Enhance the architecture with new design patterns and components**, which can boost performance, in ways that you previously didn't consider because of limited time or budget. | New design and components can optimise the system, **leading to better user experience**. For example, you can use caching or adding a content delivery network component.  It can also lead to long-term cost benefits. |
| **Use monitoring tools to analyse historical trends** and to identify the flows and code implementation paths that would benefit the most from a performance optimisation effort. We recommend application performance monitoring (APM) tools and profilers for this purpose.  Identify operation hot paths and other potential bottlenecks in the system. | When you identify the recurring problematic areas, the team can **focus where gains are the highest**. |
| **Get current and stay current with technology innovations** that can improve performance.  Take advantage of the new versions released for the dependent frameworks and libraries.  Similarly, use the new features for platform resources as they're updated and patched. | Adopting new technology can often be the motivating factor to **look for opportunities to improve**.  Code that might have been slow in the past can become faster with these updates. You also want to be aware of how certain updates negatively affect performance. |

## Checklist

* Design for scalability.
* Optimise system performance.
* Implement effective resource management.
* Conduct regular performance testing.

This checklist presents a set of recommendations for you to design your workload so it can grow and meet your workload usage demand. The goal of performance is to maintain the efficiency of every interaction with a healthy system as demand increases. When you design and implement for performance, focus on the efficiency and effectiveness of cost, complexity, supporting new requirements, technical debt, reporting, and toil.

For every system, there's a limit to how much you can scale it without redesigning, introducing a workaround, or incorporating human involvement. If you don't include performance efficiency practices and consider the tradeoffs, your design is potentially at risk. Carefully consider all the points covered in the checklist to instill confidence in your system's success.

|  |  |
| --- | --- |
| Code | Recommendation |
| [PE:01](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/performance-targets) | **Define performance targets.** Performance targets should be numerical values that are tied to workload requirements. You should implement performance targets for all workload flows. |
| [PE:02](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/performance-planning) | **Conduct performance planning.** Performance planning should be done before there are predicted changes in usage patterns. Predicted changes include seasonal variations, product updates, marketing campaigns, special events, or regulatory changes. |
| [PE:03](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/select-services) | **Select the right services.** The services and product features must support your ability to reach the workload's performance targets and accommodate expected capacity changes. The selections should also weigh the benefits of using platform features or building a custom implementation. |
| [PE:04](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/collect-performance-data) | **Collect performance data.** Workload components and flows should provide automatic, continuous, and meaningful metrics and logs. Collect data at different levels of the workload, such as the application, platform, data, and operating system levels. |
| [PE:05](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/performance-test) | **Test performance.** Perform regular testing in an environment that matches the production environment. Compare results against the performance targets and the performance benchmark. |
| [PE:06](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/optimize-code) | **Optimise logic**. Use code and logic that's performant, and ensure that it offloads responsibilities to the platform. Use logic only for its intended purpose and only when necessary. |
| [PE:07](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/prioritize-critical-flows) | **Prioritise the performance of critical flows.** The allocation of workload resources and performance optimisation efforts should prioritise the flows that support the most important business processes, users, and operations. |
| [PE:08](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/optimize-data-performance) | **Optimise data usage.** Optimise data stores for their intended and actual use in the workload. |
| PE:09 | **Respond to live performance issues.** Plan how to address performance problems by incorporating clear lines of communication and responsibilities. When a problematic situation occurs, use what you learn to identify preventive measures and incorporate them in your workload. Implement methods to return to normal operations faster when similar situations occur. |
| [PE:10](https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/continuous-performance-optimize) | **Continuously optimise performance.** Focus on components that show deteriorating performance over time, such as databases and networking features. |

## Trade-offs

* Scalability vs. Complexity: Designing for scalability can add complexity to the system architecture.
* Optimisation vs. Development Time: Continuous optimisation efforts may extend development timelines.

A workload that meets its performance targets without overprovisioning is efficient. Key strategies for performance efficiency include proper use of code optimisations, design patterns, and capacity planning. Clear performance targets and testing underpin this pillar.

During the design phase of a workload, it's important to consider how decisions based on the Performance Efficiency design principles and recommendations in the Design review checklist for Performance Efficiency might influence the goals and optimisation efforts of other pillars. Certain decisions may benefit some pillars, yet represent tradeoffs for others. This section lists example tradeoffs that a workload team might encounter when designing workload architecture and operations for performance efficiency.

**Performance Efficiency tradeoffs with Reliability**

**A cornerstone of reliability is ensuring resilience by using replication and limiting the blast radius of malfunctions.**

* Consolidating workload resources can use excess capacity and improve efficiency. However, it increases the blast radius of a malfunction in the co-located component or application platform.

**Reliability prioritises simplicity.**

* Data partitioning and sharding help avoid performance issues in large or frequently accessed datasets. However, the implementation of these patterns increases complexity because (eventual) consistency needs to be maintained across additional resources.
* Denormalising data for optimised access patterns can improve performance, but it introduces complexity because multiple representations of data need to be kept synchronised.
* Performance-centric cloud design patterns sometimes necessitate the introduction of additional components. The use of these components increases the surface area of the workload. The components then must themselves be made reliable to keep the whole workload reliable.

**Avoiding the unnecessary use of production systems is a self-preservation approach for reliability.**

* Performance testing in active environments carries the risk of causing malfunctions due to the test actions or configurations.
* Workloads should be instrumented with an application performance monitoring (APM) system that enables teams to learn from active environments. The APM tooling is installed and configured in application code or in the hosting environment. Improper use, exceeding limitations, or misconfiguration of the tool can compromise its functionality and maintenance, potentially undermining reliability.

**Performance Efficiency tradeoffs with Security**

**Security controls are established across multiple layers, sometimes redundantly, to provide defense in depth.**

One performance optimisation strategy is to remove or bypass components or processes that contribute to delays in a flow, especially when their processing time isn't justified. However, this strategy can compromise security and should be accompanied by a thorough risk analysis. Consider the following examples:

* Removing encryption in transit or at rest to improve transfer speeds exposes the data to potential integrity or confidentiality breaches.
* Removing or reducing security scanning or inspecting tools to reduce processing times can compromise the confidentiality, integrity, or availability that those tools protect.
* Removing firewall rules from network flows to improve network latency can allow undesirable communication.
* Minimising data validation for quicker data processing might compromise data integrity, especially if inputs are malicious.

**Security prioritises a reduced and contained surface area to Minimise attack vectors and reduce the management of security controls.**

Performance-centric cloud design patterns sometimes necessitate the introduction of additional components. These components increase the surface area of the workload. The new components must be secured, possibly in ways that aren't already used in the system, and they often increase the compliance scope. Consider these commonly added components:

* Introducing multiple different methods of handling business logic, like cloud flows and low-code plugins, based on the performance requirements of each task.
* Offloading processing to background jobs or even client compute.

**The Security pillar prioritises strong segmentation to enable fine-grained security controls and reduce blast radius.**

Sharing resources is an approach for improving efficiency. It increases density to optimise capacity usage. For example, re-using low-code plugins across multiple canvas apps and cloud flows. The increased density can lead to the following security concerns:

* A shared workload identity that violates the principle of least privilege and obscures individual audit trails in access logs.
* Perimeter security controls, for example network rules, that are reduced to cover all co-located components, giving individual components more access than necessary.

**Performance Efficiency tradeoffs with Operational Excellence**

**Monitoring is necessary to provide a workload with meaningful alerting and help ensure successful incident response.**

* Reducing log and metric volume to reduce the processing time spent on collecting telemetry instead of other tasks reduces the overall observability of the system. Some examples of the resulting reduced observability include:
  + It limits the data points that are used to build meaningful alerts.
  + It leads to gaps in coverage for incident response activities.
  + It limits observability in security-sensitive or compliance-sensitive interactions and boundaries.
* When performance design patterns are implemented, the complexity of the workload often increases. Components are added to critical flows. The workload monitoring strategy and performance monitoring must include those components. When a flow spans multiple components or application boundaries, the complexity of monitoring the performance of that flow increases. Flow performance needs to be correlated across all the interconnected components.

**A complex environment has more complex interactions and a higher likelihood of a negative impact from routine, ad hoc, and emergency operations.**

* Improving performance efficiency by increasing density elevates the risk in operational tasks. An error in a single process can have a large blast radius.
* As performance design patterns are implemented, they influence operational procedures like backups, key rotations, and recovery strategies. For example, data partitioning and sharding can complicate routine tasks when teams try to ensure that those tasks don't affect data consistency.

**Operational Excellence is rooted in a culture of blamelessness, respect, and continuous improvement.**

* Conducting root cause analysis of performance issues identifies deficiencies in processes or implementations that require correction. The team should consider the exercise a learning opportunity. If team members are blamed for issues, morale can be affected.
* Routine and ad hoc processes can affect workload performance. It's often considered preferable to perform these activities during off-peak hours. However, off-peak hours can be inconvenient or outside of regular hours for the team members who are responsible for or skilled in these tasks.

**Performance Efficiency tradeoffs with Experience Optimisation**

**The Experience Optimisation pillar prioritises more engaging user experiences.**

* Optimising for performance prioritises using platform features over Customisations, which deprioritise custom components that could lead to a more engaging user experience.
* Optimising for performance can focus too much on Minimising complexity, which deprioritises features for more engaging user experiences, such as custom components and integrations.
* User interface development is often done in faster iterations and ship cycles, which can make it harder to enhance performance continuously.

## Design for Scalability

Implement scalable architectures using microservices and serverless computing. Use load balancers and auto-scaling features to handle varying workloads.

## Optimise System Performance

Regularly review and optimise system performance through code tuning, query optimisation, and hardware upgrades. Use performance monitoring tools to identify bottlenecks.

## Implement Effective Resource Management

Use resource management tools to allocate and manage resources effectively. Monitor resource usage and adjust allocations based on performance requirements.

## Conduct Regular Performance Testing

Perform regular performance testing using tools like Azure Load Testing or JMeter. Use test results to identify and address performance issues promptly.

# Experience Optimisation

## Design Principles

The design principles are intended to provide guidance for aspects of Experience Optimisation that you should consider throughout the development lifecycle. Start with the recommended approaches and justify the benefits for a set of requirements. After you set your strategy, drive actions by using the Experience Optimisation checklist.

**Design for the user**

**Put the user at the center of your workload experience design.**

Strive to understand the needs, experiences, expected outcomes, and desires of the users of your workload, and tailor the design of the workload to the users' specific requirements.

|  |  |
| --- | --- |
| Approach | Benefit |
| Learn about what users want, need, and struggle with through user research and empathy-building methods. Imagine yourself as a user to understand their experiences and views, guiding design choices that meet their needs well. | Insights inform design decisions and ensure that the workload addresses users' real-world challenges effectively.  Workload experiences are tailored to users' workflows, mental models, and preferences, resulting in more relevant and user-centric designs.  The workload resonates with users on an emotional level, leading to higher levels of user satisfaction and engagement.  You find creative and innovative solutions that address users' needs and overcome usability barriers. |
| Include users in every stage of the design process, making users the main drivers of decision-making. Design solutions that match user objectives, processes, and ways of thinking, making sure that users can perform tasks quickly and easily. | Users are more likely to enjoy using the system and see its benefits if they feel that the system meets their needs well and is not difficult to use.  When users perceive the workload as user-friendly and useful for their needs, they're more likely to adopt it and use it regularly in their daily work. This perception results in higher levels of user engagement and usage, contributing to the success of the system.  Experiences that are easy to use Minimise the need for extensive training and support. When users can easily comprehend and operate the system, you can save time and resources on training programs and user support. |
| Design experiences that are inclusive and suit users with different backgrounds, cultures, and contexts. Think about factors like language, cultural expectations, and digital skills, to ensure that interfaces are easy to use and meaningful for everyone. | Your workload can be used by users with different abilities, including users with disabilities.  You anticipate and accommodate evolving user needs and technologies with a flexible and adaptable experience. |
| Maintain consistency in design and functionality across different platforms and devices. Design interfaces that adapt seamlessly to different screen sizes, resolutions, and input methods, providing users with a consistent experience regardless of the device they use. | Users have a consistent and functional experience with your workload across their different devices and platforms. |

**Design for simplicity**

**Make the workload easy to use. Minimise complexity and ensure a user-friendly experience.**

Simplicity helps users learn the system, perform their tasks, and achieve their goals with minimal distractions.

|  |  |
| --- | --- |
| Approach | Benefit |
| Create a coherent and logical structure of information, content, and functionality in the interface. | Users quickly understand the interface and are able to locate the functionality relevant to their tasks.  Users can process information more efficiently and make informed decisions without being overwhelmed.  Streamlined interactions are more accessible and ensure users can navigate the interface more effectively. |
| Provide an intuitive and consistent interface that follows relevant design conventions and standards. Leverage the existing mental models and expectations of the users. | Users can easily figure out how to use the workload without spending excessive time or effort in learning new interactions or workflows. This faster learning speeds up user onboarding and adoption.  Users have a feeling of consistency as they experience design elements and interactions they recognise and have used before. This consistency lowers mental effort and makes it simpler for users to comprehend and navigate the interface.  There is uniformity across interfaces, both within a workload and across different workloads. Uniform design elements and interactions facilitate the learning and usage of the interface for users, regardless of their prior experience. |
| Focus on the most important information and actions. Eliminate any elements, features, or fields that aren't needed. | Users access and consume information effectively, resulting in better-informed interactions and improved satisfaction.  Users find the system easy to use and are more likely to engage with the workload, leading to higher adoption rates and better utilisation. |

**Design for efficiency**

**Reduce friction and streamline the user experience.**

Design the workload to make it easy for users to accomplish tasks efficiently, with minimal frustration. Ensuring users are able to get their work done increases their satisfaction and realises higher return on investment (ROI).

|  |  |
| --- | --- |
| Approach | Benefit |
| Design workflows and processes to be straightforward and intuitive. Minimise the number of steps required to complete tasks. | Users accomplish more in less time.  The risk of errors is reduced, leading to improved accuracy and reliability.  Operational expenses are lowered through efficiency improvements. |
| Optimise interaction design to reduce the time and effort required for users to perform common actions, such as filling out forms or making selections. | Users complete the objectives more quickly.  Task completion rates go up and the need for user support goes down.  Users feel a greater sense of satisfaction and accomplishment. |
| Provide clear and consistent feedback. Inform users about the status, results, or errors of their actions; use progress indicators for long-running tasks; and acknowledge user input promptly. | Users understand what is happening, monitor their progress, and recover from errors.  Users know that the system has seen and handled their actions, making them more trustful of the interface.  Users feel more at ease when they get instant feedback, knowing that their input has been accepted.  Users move through complex processes or new interfaces with assurance.  Error messages provide instructions or suggestions for corrective actions, helping users troubleshoot problems efficiently. |
| Prioritise tasks and features based on user needs and frequency of use, ensuring that commonly performed actions are easily accessible and efficient to execute. | Task prioritisation helps users concentrate on the most important and effective activities, Minimising interruptions and mental stress. Clear prioritisation directs users to the actions that have the most impact, simplifying their workflow and improving their overall experience.  Putting the most important tasks first helps users see more clearly the actions they need to take. Users can choose faster and with more confidence, leading to less time spent on decisions and better usability.  Workload teams can use resources wisely by working on tasks or features that benefit users the most. Workload teams can increase the effect of their efforts and resources, leading to better results and ROI. |

**Negotiate realistic performance targets**

**The intended user experience is defined, and there's a strategy to develop a benchmark and measure targets against the pre-established business requirements.**

Start your design process with clear performance targets based on the business needs and expectations. Rather than just focusing on technical metrics, work with business stakeholders to set expectations and define targets that are aligned with the expected user experience of the workload.

**Design to meet performance requirements**

**Select the right services to meet performance targets.**

It's important to proactively measure performance. Choose services and features across the technology stack that enable you to meet your performance goals, monitor performance, and track which components of the workload might pose challenges. Also, define a process for testing performance.

**Achieve and sustain performance**

**Protect against performance degradation while the system is in use and as it evolves.**

Performance isn't a one-time activity. You need to keep working on it throughout the development of the workload. Expect to test and optimise your workload multiple times. Any changes to requirements, configuration, code, or product features can affect performance.

**Improve efficiency through optimisation**

**Improve system efficiency within the defined performance targets to increase workload value.**

Establish a performance culture that allows developers to spend time on performance optimisation. Adjust performance targets based on user experience, and monitor which components of your workload experience heavy load. Continue to evaluate new product features that could improve performance. The cycle of monitoring, optimising, testing, and deploying is a continuous process.

## Checklist

This checklist presents a set of recommendations to help you optimise experiences for users.

|  |  |
| --- | --- |
| Code | Recommendation |
| XO:01 | **Design your workload to meet the expectations and requirements of users**. Ensure that the workload is useful and provides a positive user experience. |
| [XO:02](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/design-standards) | **Follow established standards, conventions, and guidelines**. Leverage commonly-used design patterns. Maintain consistency in design elements, terminology, and interactions across the interface. Use consistent patterns and standards to guide users through the interface and create a cohesive user experience. |
| [XO:03](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/information-architecture) | **Implement a consistent information architecture**. Be consistent in navigation structures. Provide contextual clues and visual indicators to guide users. Use consistent and user-friendly labels for categories, navigation, and other informational elements. |
| [XO:04](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/usability) | **Prioritise ease of use during the design process**. Minimise user effort and maximise task efficiency. Streamline complex processes. Align design decisions with user needs. |
| [XO:05](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/feedback-guidance) | **Provide meaningful, useful, and simple guidance** in notifications and messages. Notify users with relevant, important, and valuable information. Give users feedback on their actions and let them know when something happens and what they need to do next. |
| [XO:06](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/layout) | **Optimise for different contexts and devices.** Maintain usable and visually appealing layouts across screen sizes and resolutions. Use adaptive techniques to dynamically render content in various ways. |
| [XO:07](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/visual-design) | **Optimise user perception and aesthetics.** Create visually appealing and engaging experiences that resonate with workload users. |
| [XO:08](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/interaction-design) | **Follow interaction design best practices.** Ensure seamless communication between users and the workload. Guide users through tasks effectively. Help users maintain context. |
| [XO:09](https://learn.microsoft.com/en-us/power-platform/well-architected/experience-optimization/user-interface-content) | **Write task-focused content in a professional tone.** Content is the primary mode of communication and should provide clear direction. Use an approachable, professional tone that provides meaningful content that facilitates success. |

## Trade-offs

During the design phase of a workload, it's important to consider how decisions based on the Experience Optimisation design principles and recommendations in the Design review checklist for Experience Optimisation might influence the goals and optimisation efforts of other pillars. Certain decisions may benefit some pillars, while being tradeoffs for others. This section lists example tradeoffs that a workload team might encounter when designing workload architecture and operations for experience optimisation.

**Experience Optimisation tradeoffs with Reliability**

**A workload that uses straightforward approaches and limits customisation is generally easier to manage in terms of reliability.**

* Continuous improvement through adding new features in the user experience can introduce new issues that affect reliability and will require thorough testing.
* Customising the user interface with code and components adds new reliability targets.

**Experience Optimisation tradeoffs with Security**

**The Security pillar prioritises a reduced and contained surface area to Minimise attack vectors and the management of security controls.**

* Improving the experience with features and integrations can introduce potential vectors for attack.
* Using code components can inadvertently increase the workload's attack surface.

**Experience Optimisation tradeoffs with Operational Excellence**

**A workload's operational process provides rigor, consistency, specificity, and prioritisation to change management in a workload.**

* Customisations in the workload's experience can complicate maintenance efforts and require increased resource utilisation for testing.
* Allocating resources to perform user research and develop prototypes can impact the focus on operational activities.

**Experience Optimisation tradeoffs with Performance Efficiency**

**Performance Efficiency prioritises platform features over customisation to achieve efficiency and effectiveness.**

* Customisations in the workload's experience can have a negative impact on performance.

# Cost Management

## Design Principles

* **Optimisation**: Continuously optimise resources to reduce costs without compromising performance.
* **Visibility**: Maintain visibility into all cost-related aspects of the system.
* **Accountability**: Assign cost accountability to specific teams or individuals.

Architecture design is always driven by business goals and must **factor in return on investment (ROI) and financial constraints**. Typical questions to consider include:

* Do the allocated budgets enable you to meet your goals?
* What's the spending pattern for the application and its operations? What are priority areas?
* How will you maximise the investment in resources, by better utilisation or by reduction?

A cost-optimised workload isn't necessarily a low-cost workload. There are significant tradeoffs. Tactical approaches are reactive and can reduce costs only in the short term. To achieve long-term financial responsibility, you need to **create a strategy with prioritisation, continuous monitoring, and repeatable processes** that focuses on optimisation.

The design principles are intended to provide optimisation strategies that you need to consider when you design and implement your workload architecture. Start with the recommended approaches and **justify the benefits for a set of business requirements**. After you set your strategy, drive actions by using the Cost Optimisation checklist as your next step.

As you prioritise business requirements to align with technology needs, you can adjust costs. However, you should expect a series of **tradeoffs in areas in which you want to optimise cost, such as security, scalability, resilience, and operability**. If the cost of addressing the challenges in those areas is high and these principles aren't applied properly, you might make risky choices in favor of a cheaper solution, ultimately affecting your business goals and reputation.

**Develop cost-management discipline**

**Build a team culture that has awareness of budget, expenses, reporting, and cost tracking.**

Cost Optimisation is conducted at various levels of the organisation. It's important to understand how your workload is aligned with organisational goals and FinOps practices. A view into the business units, resource organisation, and centralised audit policies allows you to adopt a Standardised financial system.

|  |  |
| --- | --- |
| Approach | Benefit |
| Develop a **cost model**. This fundamental exercise is a prerequisite to setting up a financial tracking system. | A cost model helps segment expenses and estimate and **forecast the total cost of ownership**, including infrastructure, support, and implementation. It enables you to identify cost drivers early and predict how any change, growth, or shrinkage will affect overall spending in your projected business model. |
| Have an **effective but flexible accountability model** that's implemented with properly assigned roles and responsibilities. | As the architecture evolves, various roles participate in decision making. Clear accountability helps **enforce the functional expectations** of each role (given a scope), drive clarity, and generate reports with transparency at desired levels. |
| Estimate **realistic budgets** that cover all non-negotiable functional and nonfunctional requirements, personnel and training costs, and processes that provide for anticipated growth. | You'll be able to **set financial boundaries** and establish ways to check your spending against the allocated budget. You'll also get notifications when certain thresholds are exceeded, which prevents overspending at the tenant scope, resource scope, and other scopes that are applied to the budget. |
| Use **governance** and processes to implement the accountability model and budgets. | It's not enough to get notifications, because that's reactionary. **Proactive governance** can help you avoid actions that might lead to unnecessary expenditure that's beyond the budget.  Certain actions can improve the current state. Are retention policies too relaxed? Do you need scalability limits to ensure responsible engineering? |
| Build capabilities in the system that **capture and classify expense**. | You'll be able to calculate the costs that **reveal technical and business perspectives** at different billing boundaries.  You'll also be able to conduct regular reviews and drive showback and chargeback processes. |
| Plan on **training costs, hiring expenses, and the cost of infrastructure needed** to augment skills as the workload matures. | Investing in staffing **complements existing skills** through full-time or vendor support. |
| Encourage **upstream communication** from architects and application owners. | Research costs are reduced when you act on feedback, which should be considered as meaningful as numeric data. You'll empower employees by using their input to **drive realistic design changes** and business strategies. |

**Design with a cost-efficiency mindset**

**Spend only on what you need to achieve the highest return on your investments.**

Every architectural decision has direct and indirect financial implications. **Understand the costs associated with build versus buy options**, technology choices, the billing model and licensing, training, operations, and so on.

Given a set of requirements, optimise and make tradeoff decisions, in relation to costs, that still effectively address the cross-cutting concerns of the workload.

|  |  |
| --- | --- |
| Approach | Benefit |
| **Measure the total cost** incurred by technology and automation choices, taking into account the impact on ROI. The design must work within the acceptable boundaries for all functional and nonfunctional requirements. The design must also be flexible to accommodate predicted evolution.  Factor in the cost of acquisition, training, and change management. | Implementing a balanced approach that takes ROI into account **prevents overengineering**, which might increase costs.  Discarding alternatives that are expensive and lack business justification provides buffer in your budget that you can spend in other areas.  We don't recommend that you design beyond planned growth because doing so might divert investments that are allocated for near-term design choices and tradeoff compensation. |
| **Establish the initial cost**, using the billing models that are best suited to fulfill your requirements. | Refining cost estimates will help you forecast how costs compare to the budget and identify the main cost drivers. Do the cost drivers help meet the business requirements?  You need to know the initial cost before you can readjust your choices and evaluate other cost-effective options. You'll uncover hidden costs that might go undetected if the design was in a purely hypothetical state. |
| **Fine-tune the design by prioritising services** that can reduce the overall cost, don't need additional investment, or don't have a significant impact on functionality. Prioritisation should account for the business model and technology choices that bring high ROI. | You'll be able to explore cheaper options that might enable resource flexibility or dynamic scaling, or you might justify the use of existing investments. The prioritisation parameters might factor in costs that are required for critical workloads, runtime, and operations, and other costs that might help the team work more efficiently. |
| **Design your architecture to support cost guardrails**. | Enforcement via governance policies or built-in application design patterns can prevent incidental or unapproved charges. |
| For workloads that are backed by service-level agreements (SLAs), **weigh the pros and cons of reserving budget for penalties versus using it for implementation**. You can avoid penalties if your implementation is sound. | Ensuring that your design fulfills its intended function and meets commitments is a proactive approach that reduces eventual risks of liability.  Negotiating realistic cost commitments or working with your product owner to create a dedicated violation budget makes these goals more achievable. |

**Design for usage Optimisation**

**Maximise the use of resources and operations. Apply them to the negotiated functional and nonfunctional requirements of the solution.**

Services and offerings provide various capabilities and pricing tiers. **After you purchase a set of features, avoid underutilising them**. Find ways to maximise your investment in the tier. Likewise, continuously evaluate billing models to find those that better align to your usage, based on current production workloads.

|  |  |
| --- | --- |
| Approach | Benefit |
| Evaluate whether your chosen **resource SKUs provide** additional features that can help you meet performance, security, reliability, or operational targets. | By taking advantage of features offered by the SKU that you selected for your design, you can maximise the use of what you paid for and **avoid paying for unused features**. |
| **Use consumption-based pricing** when it's practical. | You'll pay for exactly what you use. This option might be more expensive than a fully utilised prepaid option. However, **if you don't expect to fully utilise pre-purchased compute**, consumption billing might be a better choice. |
| **Apply policies** to comply with the design and the design's upper and lower limits. | Governance ensures that only allowed regions and services and their budgeted quantity are provisioned. This governance **reduces waste and the over-provisioning of resources**. |
| **Prioritise deployment of active-active models** or active-only over active-passive models, as part of your recovery plan, if you already paid for the resources. | If your design defaults to using active-passive models, you might have **idle resources** that could otherwise be used. Converting to active-active might enable you to meet your load leveling and scale bursting requirements without overspending. If you can meet your recovery targets with an active-only model, the costs of those resources can be removed completely. |
| Regularly and rigorously **review deployments for unused resources** and data and decommission them. | Shutting down unused resources and deleting data when you no longer need it reduces waste and **frees up funds so you can invest them elsewhere**. |
| Find additional **uses for resources that you committed to** in discounted longer-term plans. | Consider **pre-purchased resources, existing licenses, and other commitment-based discounted resources that are unused**. You can save money by using these resources. You can use these resources for tests, additional environments, or even addressing functional and nonfunctional requirements.  Likewise, finding opportunities to utilise committed plans for resources that your workload is using will enable your workload to optimise those resource costs via the precommitment. |
| Take advantage of your **investment in your support plan**. | Using your support plan to **handle production problems or for proactive reviews** will help you get your money's worth. Fully engage with your Microsoft support model. |

**Design for rate Optimisation**

**Increase efficiency without redesigning, renegotiating, or sacrificing functional or nonfunctional requirements.**

Take advantage of opportunities to optimise the utility and costs of your existing resources and operations. If you don't, you unnecessarily spend money without any added ROI.

|  |  |
| --- | --- |
| Approach | Benefit |
| Optimise by committing and pre-purchasing to **take advantage of discounts** offered on resource types that aren't expected to change over time and for which costs and utilisation are predictable.  Also, work with your licensing team to influence future purchase agreement programs and renewals. | Microsoft offers reduced rates for predictable and long-term commitment to specific resources and resource categories. Resources **cost less during the usage period** and can be amortised over the period.  By keeping your licensing team aware of the current and predicted investment by resource, you can help them **right-size commitments** when your organisation signs the agreement. In some cases, these projections and commitments could influence your organisation's price sheet, which benefits your workload's cost and also other teams that use the same technology. |
| Find ways to reduce licensing costs by evaluating **alternatives that don't require additional licensing**. Consider options like hybrid use and pre-production subscription pricing. | You'll be able to **reduce licensing costs** for services, operating systems, and tools by taking advantage of options that give you usage rights to the same or comparable technologies at a lower cost. |
| Switch to **fixed-price billing** instead of consumption-based billing for a resource when its utilisation is high and predictable and a comparable **SKU** or **billing option is available**. | When **utilisation is high and predictable**, the fixed-price model usually costs less and often supports more features. Using it could increase your ROI. |
| **Use centralised resources** that are provided by your organisation, and share the cost with other teams. | Shared resources often have higher capacity to support multiple workloads, and **costs are distributed across teams**. Taking a dependency on shared resources can save money, as long as the functionality of your workload isn't compromised.  Showback and chargeback are other potential benefits. |
| **Deploy to regions** that cost less. | Some regions offer services at a cheaper price. If you can still meet functional and nonfunctional requirements, you should consider using those regions. You can further benefit by evaluating the regional choice per environment, potentially using favorable pricing for preproduction environments even if the production environment cannot. |
| **Co-locate usage** with other resources, workloads, and even teams.  **Prefer services that make it easier to achieve higher density.**  Consider the potential tradeoffs, especially on security boundaries. | You'll be able to save costs by optimising hardware utilisation.  **As density increases, the amount of resources that you need to run a workload decreases**. This decrease reduces cost per unit and the cost of management. |

**Monitor and Optimise over time**

**Continuously right-size investment as your workload evolves with the ecosystem.**

What was important yesterday might not be important today. As you learn through evaluation of production workloads, **expect changes in architecture, business requirements, processes, and even team structure**. Your software development lifecycle (SDLC) practices might need to evolve. External factors might also change, like the cloud platform, its resources, and your agreements.

You should carefully assess the impact of all changes on cost. Monitor changes and the ROI trend on a regular cadence, and evaluate whether you need to adjust functional and nonfunctional requirements.

|  |  |
| --- | --- |
| Approach | Benefit |
| By using your cost tracking system, continuously evaluate and optimise the costs of resources, data, and paid support. Are there **underutilised resources that can be retired, replaced, rebuilt, or refactored**? | You'll reduce costs by **avoiding paying for resources that aren't fully utilised**. Understanding pricing metrics can help you make decisions that are more aligned with your cost model. It can also prevent unwarranted billing. By resizing or removing underutilised resources, or even changing SKUs, you can reduce costs.  You might also be able to save some costs by evaluating the use of your support contract and right-sizing it. |
| **Continuously adjust architecture** design decisions, resources, code, and workflows based on ROI data. | Regular reviews of metrics, performance data, billing reports, and feature usage might lead to **fine-tuning that can reduce costs**. |
| **Treat different SDLC environments differently**, and deploy the right number of environments.  Production environments should be your main cost driver. | You can save money by understanding that **not all environments need to simulate production**. Nonproduction environments can have different features, SKUs, instance counts, and even logging.  You also can save costs by creating pre-production environments on-demand and removing them when you no longer need them. |

## Checklist

* Implement cost tracking mechanisms.
* Optimise resource usage.
* Set cost reduction targets.
* Review and adjust budgets regularly.

This checklist presents a set of recommendations about cost optimisation for your workload to help you achieve a high return on investment (ROI) based on the business value that your workload delivers. Cost Optimisation balances actual costs versus perceived value, team efficiency, focus, and effort, while meeting the defined functional and nonfunctional requirements of your workload.

Every workload has direct and indirect costs, and every workload is designed to deliver value. If you don't incorporate the recommendations in this section and consider the tradeoffs, your design might not make the best use of your time and money. Carefully consider the points covered in the following checklist to instill confidence in your design's success.

Cost Optimisation is a continuous process in which you optimise workload costs and align your workload with the broader governance discipline of cost management. What's important today might not be important tomorrow. Technology choices or options and features that your platform offers today might be different. Learn from production and nonproduction environments, be aware of platform changes, and apply your findings to your workload and your workload's dependencies.

|  |  |
| --- | --- |
| Code | Recommendation |
| CO:01 | **Create a culture of financial responsibility.** Regularly train personnel so technical skills remain sharp. Foster creativity and spending accountability in the work environment. Invest in tooling and implementing automation. |
| [CO:02](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/cost-model) | **Create and maintain a cost model.** A cost model should estimate the initial cost, run rates, and ongoing costs. Negotiate a budget that covers a cost model and has a buffer for unplanned spending. |
| [CO:03](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/collect-review-cost-data) | **Collect and review cost data.** Data collection should capture daily costs. In cost reports, include incurred costs (metered), prepaid costs (amortised), trends, and forecasts. Stakeholders should regularly review spending against the budget and cost model. Automate alerts to trigger notifications at key thresholds and detect anomalies to indicate deviations from trend baselines. |
| [CO:04](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/set-spending-guardrails) | **Set spending guardrails.** Guardrails should include release gates, governance policies, resource limits, and access controls. Prioritise platform automation over manual processes. |
| [CO:05](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/get-best-rates) | **Get the best rates from providers.** You should find and use the best rates for cloud resources and licenses. Regularly review cost savings. Cost reviews should include regional pricing, pricing tiers, pricing models (consumption or commitment-based), license portability, corporate purchasing plans, and price sheets. |
| [CO:06](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/align-usage-to-billing-increments) | **Align usage to billing increments.** You should understand billing increments (meters) and align resource usage to those increments. Modify the service to align with billing increments, or modify resource usage to align with billing increments. Consider using a proof-of-concept to validate billing knowledge and design choices for major cost drivers and to reveal ways to align billing and resource usage. |
| [CO:07](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-component-costs) | **Optimise component costs.** Regularly remove or optimise legacy, unneeded, and underutilised workload components, including application features, platform features, and resources. |
| [CO:08](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-environment-costs) | **Optimise environment costs.** Align spending to Prioritise preproduction, production, operations, and disaster recovery environments. For each environment, consider the required availability, licensing, operating hours and conditions, and security. Nonproduction environments should emulate the production environment. Implement strategic tradeoffs into nonproduction environments. |
| [CO:09](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-flow-costs) | **Optimise flow costs.** Align the cost of each flow with flow priority. When you Prioritise flows, consider the features, functionality, and nonfunctional requirements of each flow. Optimising flow spend often requires strategic compromises. |
| [CO:10](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-data-costs) | **Optimise data costs.** Data spending with data priority. Data optimisation should include improvements to data management (tiering and retention), volume, replication, backups, file formats, and storage solutions. |
| [CO:11](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-code-costs) | **Optimise code costs.** Evaluate and modify code to meet functional and nonfunctional requirements with fewer or cheaper resources. |
| [CO:12](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-scaling-costs) | **Optimise scaling costs.** Evaluate alternative scaling within your scale units. Consider alternative scaling configurations, and align with the cost model. Considerations should include utilisation against the inherit limits of every instance, resource, and scale unit boundary. Use strategies for controlling demand and supply. |
| [CO:13](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/optimize-personnel-time) | **Optimise personnel time.** Align the time personnel spends on tasks with the priority of the task. The goal is to reduce the time spent on tasks without degrading the outcome. Optimisation efforts should include Minimising noise, reducing build times, high fidelity debugging, and production mocking. |
| [CO:14](https://learn.microsoft.com/en-us/azure/well-architected/cost-optimization/consolidation) | **Consolidate resources and responsibility.** Look within the workload for ways to consolidate resources and increase density. Outside the workload, use existing centralised resources and services that enable you to consolidate workload responsibilities. |

## Trade-offs

* **Cost vs. Performance**: Reducing costs may impact system performance if not done carefully.
* **Visibility vs. Complexity**: Increased visibility can add to system complexity, requiring additional management efforts.

When you design a workload to maximise return on investment (ROI) under financial constraints, you first need clearly defined functional and nonfunctional requirements. A work and effort Prioritisation strategy is essential. The foundation is a team that has a strong sense of financial responsibility. The team should have a strong understanding of available technologies and their billing models.

After you understand the ROI of a workload, you can start improving it. To improve the ROI, consider how decisions based on the Cost Optimisation design principles and the recommendations in the design review checklist for Cost Optimisation might influence the goals and optimisations of other Azure Well-Architected Framework pillars. For cost optimisation, it's important to avoid focusing on a cheaper solution. Choices that focus only on Minimising spending can increase the risk of undermining your workload's business goals and reputation. This section describes example tradeoffs that a workload team might encounter when considering the target setting, design, and operations for cost optimisation.

**Cost Optimisation tradeoffs with Reliability**

The cost of a service disruption must be measured against the cost of preventing or recovering from one. If the cost of disruptions exceeds the cost of reliability design, you should invest more to prevent or mitigate disruptions. Conversely, the cost of the reliability efforts might be more than the cost of a disruption, including factors like compliance requirements and reputation. You should consider strategic divestment in reliability design only in this scenario.

**A workload incorporates resiliency measures to attempt to avoid and withstand specific types and quantities of malfunction.**

* To save money, the workload team might underprovision a component or overconstrain its scaling, making the component more likely to fail during sudden spikes in demand.
* Consolidating workload resources (*increasing density*) for cost optimisation makes individual components more likely to fail during spikes in demand and during maintenance operations like updates.
* Removing components that support resiliency design patterns, like a message bus, and creating a direct dependency reduces self-preservation capabilities.
* Saving money by reducing redundancy can limit a workload's ability to handle concurrent malfunctions.
* Using budget SKUs might limit the maximum service-level objective (SLO) that the workload can reach.
* Setting hard spending limits can prevent a workload from scaling to meet legitimate demand.
* Without reliability testing tools or tests, the reliability of a workload is unknown, and it's less likely to meet reliability targets.

**A workload that's reliable has a tested incident response and recovery plan for disaster scenarios.**

* Reduced testing or drilling of a workload's disaster recovery plan might affect the speed and effectiveness of recovery operations.
* Creating or retaining fewer backups decreases possible recovery points and increases the chance of losing data.
* A less expensive support contract might increase workload recovery time due to potential delays in technical assistance.

**A workload that uses straightforward approaches and avoids unnecessary or overengineered complexity is generally easier to manage in terms of reliability.**

* Using cost-optimisation cloud patterns can add new components, like a content delivery network (CDN), or shift duties to edge and client devices that a workload must provide reliability targets for.
* Event-based scaling can be more complicated to tune and validate than resource-based scaling.
* Reducing data volume and tiering data through data lifecycle actions, possibly in conjunction with implementing aggregated data points before a lifecycle event, introduces reliability factors to consider in the workload.
* Using different regions to optimise cost can make management, networking, and monitoring more difficult.
* **Cost Optimisation tradeoffs with Security**

The cost of a compromise to confidentiality, integrity, and availability in a workload must always be balanced against the cost of the effort to prevent that compromise. A security incident can have a wide range of financial and legal impacts and harm a company's reputation. Investing in security is a risk mitigation activity. The cost of experiencing the risks must be balanced against the investment. As a rule, don't compromise on security to gain cost optimisations that are below the point of responsible and agreed upon risk mitigation. Optimising security costs by rightsizing solutions is an important optimisation practice, but be aware of tradeoffs like the following when doing so.

**Security controls are established across multiple layers, sometimes redundantly, to provide defense in depth.**

One cost optimisation tactic is to look for ways to remove components or processes that accrue unit or operational costs. Be aware that removing security components like the following examples for the sake of saving money impacts security. You need to carefully perform a risk analysis on this impact.

* Reducing or simplifying authentication and authorisation techniques compromises the *verify explicitly* principle of zero-trust architecture. Examples of these simplifications include using a basic authentication scheme like preshared keys rather than investing time to learn industry OAuth approaches, or using simplified role-based access control assignments to reduce management overhead.
* Removing encryption in transit or at rest to reduce costs on certificates and their operational processes exposes data to potential integrity or confidentiality breaches.
* Removing or reducing security scanning or inspection tooling or security testing because of the associated cost and time investment can directly impact the confidentiality, integrity, or availability that the tooling and testing is intended to protect.
* Reducing the frequency of security patching because of the operational time invested in cataloging and performing the patching affects a workload's ability to address evolving threats.
* Removing network controls like firewalls might lead to a failure to block malicious inbound and outbound traffic.

**The Security pillar Prioritises a reduced and contained surface area to Minimise attack vectors and the management of security controls.**

Cloud design patterns that optimise costs sometimes necessitate the introduction of additional components. These additional components increase the surface area of the workload. The components and the data within them must be secured, possibly in ways that aren't already used in the system. These components and data are often subject to compliance. Examples of patterns that can introduce components include:

* Using the Static Content Hosting pattern to offload data to a new CDN component.
* Using the Valet Key pattern to offload processing and secure resource access to client compute.
* Using the Queue-Based Load Leveling pattern to smooth costs by introducing a message bus.

**The Security pillar Prioritises strong segmentation to support the application of targeted security controls and to control the blast radius.**

Sharing resources, for example in multi-tenancy situations or co-locating multiple applications on a shared application platform, is an approach for reducing costs by increasing density and reducing the management surface. This increased density can lead to security concerns like these:

* Lateral movement between components that share resources is easier. A security event that compromises the availability of the application platform host or an individual application also has a larger blast radius.
* Co-located resources might share a workload identity and have less meaningful audit trails in access logs.
* Network security controls must be broad enough to cover all co-located resources. This configuration potentially violates the principle of least privilege for some resources.
* Co-locating disparate applications or data on a shared host can lead to extending compliance requirements and security controls to applications or data that would otherwise be out of scope. This broadening of scope necessitates additional security scrutiny and auditing effort on the co-located components.
* **Cost Optimisation tradeoffs with Operational Excellence**

**A workload's SDLC process provides rigor, consistency, specificity, and Prioritisation to change management in a workload.**

* Reducing testing efforts to save time and the cost associated with test personnel, resources, and tooling can result in more bugs in production.
* Delaying paying back technical debt to focus personnel efforts on new features can lead to slower development cycles and overall reduced agility.
* DePrioritising documentation to focus personnel efforts on product development can lead to longer onboarding time for new employees, impact the effectiveness of incident response, and compromise compliance requirements.
* A lack of investment in training leads to stagnated skills, reducing the team's ability to adopt newer technologies and practices.
* Removing automation tooling to save money can result in personnel spending more time on the tasks that are no longer automated. It also increases the risk of errors and inconsistencies.
* Reducing planning efforts, like scoping and activity Prioritisation, to cut expenses can increase the likelihood of rework due to vague specifications and poor implementation.
* Avoiding or reducing continuous improvement activities, like retrospectives and after-incident reports, to keep the workload team focused on delivery can create missed opportunities to optimise routine, unplanned, and emergency processes.

**Observability is necessary to help ensure that a workload has meaningful alerting and successful incident response.**

* Decreasing log and metric volume to save on storage and transfer costs reduces system observability and can lead to:
  + Fewer data points for creating alerts related to reliability, security, and performance.
  + Coverage gaps in incident response activities.
  + Limited observability into interactions or boundaries related to security or compliance.
* Cost Optimisation design patterns can add components to a workload, increasing its complexity. The workload monitoring strategy must include those new components. For example, some patterns might introduce flows that span multiple components or shift processes from the server to the client. These changes can increase the complexity of correlating and tracking information.
* Reduced investment in observability tooling and the maintenance of effective dashboards can decrease the ability to learn from production, validate design choices, and inform product design. This reduction can also hamper incident response activities and make it harder to meet the recovery time objective and SLO.

**Workload teams are expected to keep code, tooling, software packages, and operating systems patched and up to date in a timely and orderly way.**

* Letting maintenance contracts with tooling vendors expire can result in missed optimisation features, bug resolutions, and security updates.
* Increasing the time between system patches to save time can lead to missed bug fixes or a lack of protection against evolving security threats.
* **Cost Optimisation tradeoffs with Performance Efficiency**

The Cost Optimisation and Performance Efficiency pillars both Prioritise maximising a workload's value. Performance Efficiency emphasises meeting performance targets without spending more than necessary. Cost Optimisation emphasises maximising the value produced by a workload's resources without exceeding performance targets. As a result, Cost Optimisation often improves Performance Efficiency. However, there are Performance Efficiency tradeoffs associated with Cost Optimisation. These tradeoffs can make it harder to reach performance targets and hinder ongoing performance optimisation.

**A performance-efficient workload has enough resources to serve demand but doesn't have excessive unused overhead, even when usage patterns fluctuate.**

* Reducing costs by downsizing resources can deprive applications of resources. The application might not be able to handle significant usage pattern fluctuations.
* Limiting or delaying scaling to cap or reduce costs might result in insufficient supply to meet demand.
* Autoscale settings that scale down aggressively to reduce costs might leave a service unprepared for sudden spikes in demand or cause frequent scaling fluctuations (flapping).

**Evaluating the effects of changes in functionality, changes in usage patterns, new technologies, and different approaches on the workload is one way to try to increase efficiency.**

* Limiting the focus on developing expertise in performance optimisation in order to Prioritise delivery can cause missed opportunities for improving resource usage efficiency.
* Removing access performance testing or monitoring tools increases the risk of undetected performance issues. It also limits the ability for a workload team to execute on measure/improve cycles.
* Neglecting areas prone to performance degradation, like data stores, can gradually deteriorate query performance and elevate overall system usage.

## Implement Cost Tracking Mechanisms

Use tools like Azure Cost Management to track and analyse costs associated with the development and deployment of solutions. Generate regular cost reports to maintain visibility.

## Optimise Resource Usage

Regularly review resource usage and identify opportunities for optimisation. Use auto-scaling and right-sizing to ensure resources are used efficiently.

## Set Cost Reduction Targets

Define specific cost reduction targets and monitor progress towards achieving them. Use historical data to set realistic and achievable targets.

## Review and Adjust Budgets Regularly

Conduct regular budget reviews to ensure alignment with cost management goals. Adjust budgets based on changing requirements and resource usage patterns.