Links

Friday, June 4, 2021 8:14 PM

https://docs.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-best-practices

https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga

https://dotnettutorials.net/lesson/dbcontext-entity-framework-core/

http://tutorials.jenkov.com/oauth2/roles.html

https://docs.microsoft.com/en-us/azure/architecture/framework/scalability/design-checklist

Summary

Monday, July 12, 2021 8:29 PM

- Scalability
  - App service - Inbuilt
  - Azure Fabric - VM scale set
  - API Management
  - Azure Gateway - Distribute the traffic based on avaliability Test
  - Containization/AKS - Replicat Settings.
  - Azure Functions - Inbuilt
  - Azure Logic App's - Inbuilt
- Reliability
  - Resiliency
  - FMA
  - Zone Awareness
  - Ensure connectivity
  - Scalability

- Avaliability
  - Scale Set
  - LB
  - Gateway - Distribute the traffic based on region & DNS
  - Azure Front Door service - Primary/ Secondary region/Health Prob's.
  - Azure Traffic Manager
  - Azure Load Balancer
  - Service Fabric
  - Application Insight
    - Avalaibility Test
    - Application Map
    - Diagnoistic settings
    - Smart Detection
    - Usage Analysis
- BCP
  - Traffic Manager
  - Azure Site Recovery
  - Blob storage
  - Azure Active Directory
  - VPN Gateway
  - Virtual Network
  - Geo-Replication - Azure SQL DB
- Performance
  - Azure Monitor Service
    - App Insights
- Security
  - AD

NFR

Monday, July 12, 2021 10:44 AM

- **Scaliability**
  - Azure Service Fabric - Virtual machine scale sets offer autoscale capabilities for true IaaS scenarios.
  - Azure App Gateway and Azure API Management - PaaS offerings for ingress services that enable autoscale.
  - Azure Functions, Azure Logic Apps, and App Services - Serverless pay-per-use consumption modeling that inherently provide autoscaling capabilities.
  - Azure SQL Database - PaaS platform to change performance characteristics of a database on the fly and assign more resources when needed or release the resources when they are not needed. Allows scaling up/down, read scale-out, and global scale-out/sharding capabilities

From <https://docs.microsoft.com/en-us/azure/architecture/framework/scalability/design-scale>

- Operational
  - CI/CD
    - Docker
    - Source Code control
    - Build Pipeline
    - Realse Pipleline
  - Testing
    - Unit Testing
      - Syntax corrcetness
      - Code best pratcies
    - Smoke Testing
      - correctly built
      - expected functionality
      - performance.
    - Integration Testing
      - Making sure that the different application components operate correctly individually, integration testing has as goal determine whether they can interact with each other as they should.
    - Application Manual Testing
      - Blue/Green deployments
        when deploying a new application version, you can deploy it in parallel to the existing one. This way you can start redirecting clients to the new version, and if everything goes well you will decommission the old version. If there is any problem with the new deployment, you can always redirect the users back to the old one
      - Canary **releases**
        You can expose new functionality of your application (ideally using feature flags) to a **select group of users**. If users are satisfied with the new functionality, you can extend it to the rest of the user community. In this case we are talking about releasing functionality, and not necessarily about deploying a new version of the application.

- □ A/B testing
  A/B testing is similar to canary release-testing, but while canary releases focus on mitigate risk, A/B testing focus on evaluating the effectiveness of two similar ways of achieving different goals. For example, if you have two versions of the layout of a certain area of your application, you could send half of your users to one, the other half to the other, and use some metrics to see which layout works better for the application goals.

- Business Continuity
  - Disaster Recovery - Azure site Recovery
  - DB - Geo Redency [ ]
  - DB - Auto Failover [ ]
- Avaliability
  - App Service plan
  - VM - availability set
  - Consider deploying across multiple regions
  - Redeploy to a secondary region - Azure Site Recovery.
  - Availability Zones
- Monitoring and Alerts
  - Application Insight
  - Application Map
  - Smart Detection will warn you when anomalies in performance or utilization patterns happen
  - Usage Analysis can give you telemetry on which features of your application are most frequently used, or whether all your application functionality is being used. This feature is especially useful after changes to the application functionality, to verify whether those changes were successful
  - https://docs.microsoft.com/en-us/azure/architecture/framework/devops/monitoring
- Reliability
  - Build availability targets and recovery targets into your design
    - URL ping Test
  - Ensure connectivity
    - load balancer
    - Eliminate all single points of failure from the data path (on-premises and Azure.
    - Simulate a failure path to ensure connectivity is available over alternative paths.
  - Use zone-aware services
  - Design resilience to respond to outages
  - Design for scalability
- DR
  - Traffic Manager,
  - Azure Site Recovery,
  - Azure Active Directory, VPN Gateway and Virtual Network
- DNS traffic is routed via Traffic Manager which can easily move traffic from one site to another based on policies defined by your organization.
- Azure Site Recovery orchestrates the replication of machines and manages the configuration of the failback procedures.
- Blob storage stores the replica images of all machines that are protected by Site Recovery.
- Azure Active Directory is the replica of the on-premises Azure Active Directory services allowing cloud applications to be authenticated and authorized by your company.
- VPN Gateway: The VPN gateway maintains the communication between the on

-premises network and the cloud network securely and privately.

- Virtual Network: The virtual network is where the failover site will be created when a disaster occurs.

From <https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/disaster-recovery-enterprise-scale-dr>

Monday, July 12, 2021 11:38 AM

total Experience of 12+ years including 3+ years expertise in .Net. Net Core architecture design, estimation techniques and development activities

Experience in Technical, Deployment architecture design and ability to convert the Non-functional requirements into architecture design as required

Experience in Web & Windows Applications development, maintenance experience with C#, React, Micro services, 3 tier Architecture, Web API, WCF, Web services, Entity Framework, Agile, CICD, Containerised applications

- Experience in Technical write up, Architecture diagrams, Deployment diagrams, Solution integrations and Estimations in standard frameworks (SABSA, TOGAF, NIST, 4+1 etc.)

- Experience in Structured (Sql Server, Postgres), No SQL and Hierarchical databases

- Experience in providing E2E technical solution for global customers in Microsoft stack

Preferred:

- Experience in RFP RFI process to provide a technical solution, estimation and technical write up

- Azure or Microsoft Certifications

- Application Migration and modernization - Legacy to Cloud (Tandem, Sun, AS 400 etc.)

Check List

Monday, July 12, 2021 10:46 AM

DevOps is the integration of development, quality assurance, and IT operations into a unified culture and set of processes for delivering software. Use this checklist as a starting point to assess your DevOps culture and process.

Culture

Ensure business alignment across organizations and teams. Conflicts over resources, purpose, goals, and priorities within an organization can be a risk to successful operations. Ensure that the business, development, and operations teams are all aligned.

Ensure the entire team understands the software lifecycle. Your team needs to understand the overall lifecycle of the application, and which part of the lifecycle the application is currently in. This helps all team members know what they should be doing now, and what they should be planning and preparing for in the future.

Reduce cycle time. Aim to minimize the time it takes to move from ideas to usable developed software. Limit the size and scope of individual releases to keep the test burden low. Automate the build, test, configuration, and deployment processes whenever possible. Clear any obstacles to communication among developers, and between developers and operations.

Review and improve processes. Your processes and procedures, both automated and manual, are never final. Set up regular reviews of current workflows, procedures, and documentation, with a goal of continual improvement.

Do proactive planning. Proactively plan for failure. Have processes in place to quickly identify issues when they occur, escalate to the correct team members to fix, and confirm resolution.

Learn from failures. Failures are inevitable, but it's important to learn from failures to avoid repeating them. If an operational failure occurs, triage the issue, document the cause and solution, and share any lessons that were learned. Whenever possible, update your build processes to automatically detect that kind of failure in the future.

Optimize for speed and collect data. Every planned improvement is a hypothesis. Work in the smallest increments possible. Treat new ideas as experiments. Instrument the experiments so that you can collect production data to assess their effectiveness. Be prepared to fail fast if the hypothesis is wrong.

Allow time for learning. Both failures and successes provide good opportunities for learning. Before moving on to new projects, allow enough time to gather the important lessons, and make sure those lessons are absorbed by your team. Also give the team the time to build skills, experiment, and learn about new tools and techniques.

Document operations. Document all tools, processes, and automated tasks with the same level of quality as your product code. Document the current design and architecture of any systems you support, along with recovery processes and other

maintenance procedures. Focus on the steps you actually perform, not theoretically optimal processes. Regularly review and update the documentation. For code, make sure that meaningful comments are included, especially in public APIs, and use tools to automatically generate code documentation whenever possible.

Share knowledge. Documentation is only useful if people know that it exists and can find it. Ensure the documentation is organized and easily discoverable. Be creative: Use brown bags (informal presentations), videos, or newsletters to share knowledge.

Development

Provide developers with production-like environments. If development and test environments don't match the production environment, it is hard to test and diagnose problems. Therefore, keep development and test environments as close to the production environment as possible. Make sure that test data is consistent with the data used in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

Ensure that all authorized team members can provision infrastructure and deploy the application. Setting up production-like resources and deploying the application should not involve complicated manual tasks or detailed technical knowledge of the system. Anyone with the right permissions should be able to create or deploy production-like resources without going to the operations team.

This recommendation doesn't imply that anyone can push live updates to the production deployment. It's about reducing friction for the development and QA teams to create production-like environments.

Instrument the application for insight. To understand the health of your application, you need to know how it's performing and whether it's experiencing any errors or problems. Always include instrumentation as a design requirement, and build the instrumentation into the application from the start. Instrumentation must include event logging for root cause analysis, but also telemetry and metrics to monitor the overall health and usage of the application.

Track your technical debt. In many projects, release schedules can get prioritized over code quality to one degree or another. Always keep track when this occurs. Document any shortcuts or other suboptimal implementations, and schedule time in the future to revisit these issues.

Consider pushing updates directly to production. To reduce the overall release cycle time, consider pushing properly tested code commits directly to production. Use feature toggles to control which features are enabled. This allows you to move from development to release quickly, using the toggles to enable or disable features. Toggles are also useful when performing tests such as canary releases, where a particular feature is deployed to a subset of the production environment.

Testing

Automate testing. Manually testing software is tedious and susceptible to error. Automate common testing tasks and integrate the tests into your build processes. Automated testing ensures consistent test coverage and reproducibility. Integrated UI tests should also be performed by an automated tool. Azure offers development and test resources that can help you configure and execute testing. For more information,

see Development and test.

Test for failures. If a system can't connect to a service, how does it respond? Can it recover once the service is available again? Make fault injection testing a standard part of review on test and staging environments. When your test process and practices are mature, consider running these tests in production.

Test in production. The release process doesn't end with deployment to production. Have tests in place to ensure that deployed code works as expected. For deployments that are infrequently updated, schedule production testing as a regular part of maintenance.

Automate performance testing to identify performance issues early. The impact of a serious performance issue can be as severe as a bug in the code. While automated functional tests can prevent application bugs, they might not detect performance problems. Define acceptable performance goals for metrics like latency, load times, and resource usage. Include automated performance tests in your release pipeline, to make sure the application meets those goals.

Perform capacity testing. An application might work fine under test conditions, and then have problems in production due to scale or resource limitations. Always define the maximum expected capacity and usage limits. Test to make sure the application can handle those limits, but also test what happens when those limits are exceeded. Capacity testing should be performed at regular intervals.

After the initial release, you should run performance and capacity tests whenever updates are made to production code. Use historical data to fine-tune tests and to determine what types of tests need to be performed.

Perform automated security penetration testing. Ensuring your application is secure is as important as testing any other functionality. Make automated penetration testing a standard part of the build and deployment process. Schedule regular security tests and vulnerability scanning on deployed applications, monitoring for open ports, endpoints, and attacks. Automated testing does not remove the need for in-depth security reviews at regular intervals.

Perform automated business continuity testing. Develop tests for large-scale business continuity, including backup recovery and failover. Set up automated processes to perform these tests regularly.

Release

Automate deployments. Automate deploying the application to test, staging, and production environments. Automation enables faster and more reliable deployments, and ensures consistent deployments to any supported environment. It removes the risk of human error caused by manual deployments. It also makes it easy to schedule releases for convenient times, to minimize any effects of potential downtime. Have systems in place to detect any problems during rollout, and have an automated way to roll forward fixes or roll back changes.

Use continuous integration. Continuous integration (CI) is the practice of merging all developer code into a central codebase on a regular schedule, and then automatically performing standard build and test processes. CI ensures that an entire team can work on a codebase at the same time without having conflicts. It also ensures that code

defects are found as early as possible. Preferably, the CI process should run every time that code is committed or checked in. At the very least, it should run once per day.
Consider adopting a trunk based development model. In this model, developers commit to a single branch (the trunk). There is a requirement that commits never break the build. This model facilitates CI, because all feature work is done in the trunk, and any merge conflicts are resolved when the commit happens.
Consider using continuous delivery. Continuous delivery (CD) is the practice of ensuring that code is always ready to deploy, by automatically building, testing, and deploying code to production-like environments. Adding continuous delivery to create a full CI/CD pipeline will help you detect code defects as soon as possible, and ensures that properly tested updates can be released in a very short time.
Continuous *deployment* is an additional process that automatically takes any updates that have passed through the CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning, and may not be appropriate for all teams.
Make small incremental changes. Large code changes have a greater potential to introduce bugs. Whenever possible, keep changes small. This limits the potential effects of each change, and makes it easier to understand and debug any issues.
Control exposure to changes. Make sure you're in control of when updates are visible to your end users. Consider using feature toggles to control when features are enabled for end users.
Implement release management strategies to reduce deployment risk. Deploying an application update to production always entails some risk. To minimize this risk, use strategies such as canary releases or blue-green deployments to deploy updates to a subset of users. Confirm the update works as expected, and then roll the update out to the rest of the system.
Document all changes. Minor updates and configuration changes can be a source of confusion and versioning conflict. Always keep a clear record of any changes, no matter how small. Log everything that changes, including patches applied, policy changes, and configuration changes. (Don't include sensitive data in these logs. For example, log that a credential was updated, and who made the change, but don't record the updated credentials.) The record of the changes should be visible to the entire team.
Consider making infrastructure immutable. Immutable infrastructure is the principle that you shouldn't modify infrastructure after it's deployed to production. Otherwise, you can get into a state where ad hoc changes have been applied, making it hard to know exactly what changed. Immutable infrastructure works by replacing entire servers as part of any new deployment. This allows the code and the hosting environment to be tested and deployed as a block. Once deployed, infrastructure components aren't modified until the next build and deploy cycle.

Monitoring

Make systems observable. The operations team should always have clear visibility into the health and status of a system or service. Set up external health endpoints to monitor status, and ensure that applications are coded to instrument the operations metrics. Use a common and consistent schema that helps you correlate events across systems. Azure

Diagnostics and Application Insights are the standard method of tracking the health and status of Azure resources. Azure Monitor also provides centralized monitoring and management for cloud or hybrid solutions.

Aggregate and correlate logs and metrics. A properly instrumented telemetry system will provide a large amount of raw performance data and event logs. Make sure that telemetry and log data is processed and correlated in a short period of time, so that operations staff always have an up-to-date picture of system health. Organize and display data in ways that give a cohesive view of any issues, so that whenever possible it's clear when events are related to one another.

Consult your corporate retention policy for requirements on how data is processed and how long it should be stored.

Implement automated alerts and notifications. Set up monitoring tools like Azure Monitor to detect patterns or conditions that indicate potential or current issues, and send alerts to the team members who can address the issues. Tune the alerts to avoid false positives.

Monitor assets and resources for expirations. Some resources and assets, such as certificates, expire after a given amount of time. Make sure to track which assets expire, when they expire, and what services or features depend on them. Use automated processes to monitor these assets. Notify the operations team before an asset expires, and escalate if expiration threatens to disrupt the application.

Management

Automate operations tasks. Manually handling repetitive operations processes is error-prone. Automate these tasks whenever possible to ensure consistent execution and quality. Code that implements the automation should be versioned in source control. As with any other code, automation tools must be tested.

Take an infrastructure-as-code approach to provisioning. Minimize the amount of manual configuration needed to provision resources. Instead, use scripts and Azure Resource Manager templates. Keep the scripts and templates in source control, like any other code you maintain.

Consider using containers. Containers provide a standard package-based interface for deploying applications. Using containers, an application is deployed using self-contained packages that include any software, dependencies, and files needed to run the application, which greatly simplifies the deployment process.

Containers also create an abstraction layer between the application and the underlying operating system, which provides consistency across environments. This abstraction can also isolate a container from other processes or applications running on a host.

Implement resiliency and self-healing. Resiliency is the ability of an application to recover from failures. Strategies for resiliency include retrying transient failures, and failing over to a secondary instance or even another region. For more information, see Designing reliable Azure applications . Instrument your applications so that issues are reported immediately and you can manage outages or other system failures.

Have an operations manual. An operations manual or *runbook* documents the procedures and management information needed for operations staff to maintain a system. Also document any operations scenarios and mitigation plans that might come

into play during a failure or other disruption to your service. Create this documentation during the development process, and keep it up to date afterwards. This is a living document, and should be reviewed, tested, and improved regularly.

Shared documentation is critical. Encourage team members to contribute and share knowledge. The entire team should have access to documents. Make it easy for anyone on the team to help keep documents updated.

Document on-call procedures. Make sure on-call duties, schedules, and procedures are documented and shared to all team members. Keep this information up-to-date at all times.

Document escalation procedures for third-party dependencies. If your application depends on external third-party services that you don't directly control, you must have a plan to deal with outages. Create documentation for your planned mitigation processes. Include support contacts and escalation paths.

Use configuration management. Configuration changes should be planned, visible to operations, and recorded. This could take the form of a configuration management database, or a configuration-as-code approach. Configuration should be audited regularly to ensure that what's expected is actually in place.

Get an Azure support plan and understand the process. Azure offers a number of support plans. Determine the right plan for your needs, and make sure the entire team knows how to use it. Team members should understand the details of the plan, how the support process works, and how to open a support ticket with Azure. If you are anticipating a high-scale event, Azure support can assist you with increasing your service limits. For more information, see the Azure Support FAQs.

Follow least-privilege principles when granting access to resources. Carefully manage access to resources. Access should be denied by default, unless a user is explicitly given access to a resource. Only grant a user access to what they need to complete their tasks. Track user permissions and perform regular security audits.

Use Azure role-based access control. Assigning user accounts and access to resources should not be a manual process. Use Azure role-based access control (Azure RBAC) grant access based on Azure Active Directory identities and groups.

Use a bug tracking system to track issues. Without a good way to track issues, it's easy to miss items, duplicate work, or introduce additional problems. Don't rely on informal person-to-person communication to track the status of bugs. Use a bug tracking tool to record details about problems, assign resources to address them, and provide an audit trail of progress and status.

Manage all resources in a change management system. All aspects of your DevOps process should be included in a management and versioning system, so that changes can be easily tracked and audited. This includes code, infrastructure, configuration, documentation, and scripts. Treat all these types of resources as code throughout the test/build/review process.

Use checklists. Create operations checklists to ensure processes are followed. It's common to miss something in a large manual, and following a checklist can force attention to details that might otherwise be overlooked. Maintain the checklists, and continually look for ways to automate tasks and streamline processes.

For more about DevOps, see What is DevOps? on the Visual Studio site.

From <https://docs.microsoft.com/en-us/azure/architecture/checklist/dev-ops>

Reliability Checklist

Monday, July 12, 2021 1:28 PM

LB
Scale set
Azure Traffic Manager

- Define availability and recovery targets to meet business requirements.
- Build resiliency and availability into your apps by gathering requirements.
- Ensure that application and data platforms meet your reliability requirements.
- Configure connection paths to promote availability.
- Use Availability Zones where applicable to improve reliability and optimize costs.
- Ensure that your application architecture is resilient to failures.
- Know what happens if the requirements of Service Level Agreements are not met.
- Identify possible failure points in the system to build resiliency.
- Ensure that applications can operate in the absence of their dependencies.

From <https://docs.microsoft.com/en-us/azure/architecture/framework/resiliency/design-checklist>

Operational Design Patterns

Monday, July 12, 2021 10:47 AM

attern	Summary
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
Strangler	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

From <https://docs.microsoft.com/en-us/azure/architecture/framework/devops/devops-patterns>

NFR Azure Service

Monday, July 12, 2021 1:29 PM

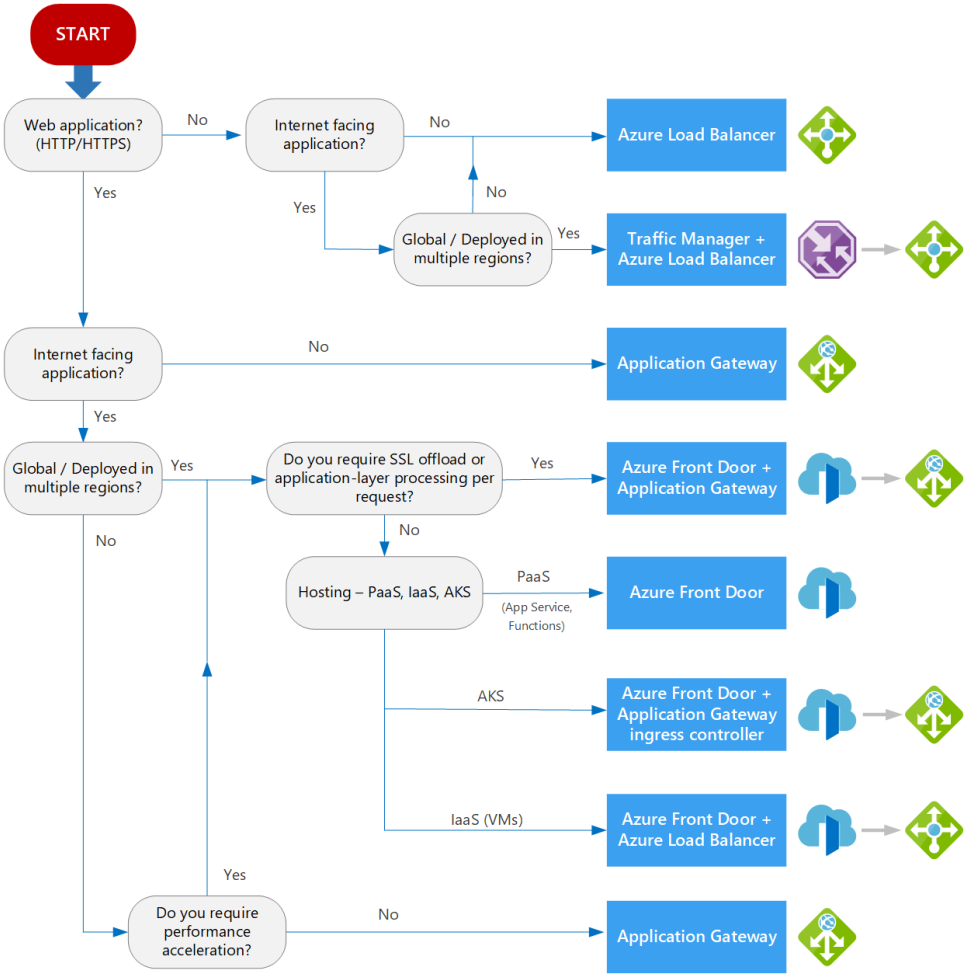
Azure Front Door - Avaliability
Azure Traffic Manager - Avaliabilty
Azure Load Balancer - Avaliability
Service Fabric - Avaliability

Kubernetes Service (AKS) - Scability

- Azure Service Fabric - Virtual machine scale sets offer autoscale capabilities for true IaaS scenarios.
- Azure App Gateway and Azure API Management - PaaS offerings for ingress services that enable autoscale.
- Azure Functions, Azure Logic Apps, and App Services - Serverless pay-per-use consumption modeling that inherently provide autoscaling capabilities.
- Azure SQL Database - PaaS platform to change performance characteristics of a database on the fly and assign more resources when needed or release the resources when they are not needed. Allows scaling up/down, read scale-out, and global scale-out/sharding capabilities.

From <https://docs.microsoft.com/en-us/azure/architecture/framework/scalability/design-scale>

Azure Site Recovery - DR


START
Web application? (HTTP/HTTPS)
No
Internet facing application?
No
Azure Load Balancer
Yes
Yes
No
Global / Deployed in multiple regions?
Yes
Traffic Manager + Azure Load Balancer
Internet facing application?
No
Application Gateway
Yes
Global / Deployed in multiple regions?
Yes
Do you require SSL offload or application-layer processing per request?
Yes
Azure Front Door + Application Gateway
No
No
Hosting – PaaS, IaaS, AKS
PaaS
(App Service, Functions)
Azure Front Door
AKS
Azure Front Door + Application Gateway ingress controller
IaaS (VMs)
Azure Front Door + Azure Load Balancer
Yes
Do you require performance acceleration?
No
Application Gateway

Pattern

Monday, July 12, 2021 11:38 AM

Operational excellence disciplines	Description
Application design	Provides guidance on how to design, build, and orchestrate workloads with DevOps principles in mind
Monitoring	Something that enterprises have been doing for years, enriched with some specifics for applications running in the cloud
Application performance management	The monitoring and management of performance and availability of software applications through DevOps
Code deployment	How you deploy your application code is going to be one of the key factors that will determine your application stability
Infrastructure provisioning	Frequently known as "Automation" or "Infrastructure as code", this discipline refers to best practices for deploying the platform where your application will run on
Testing	Testing is fundamental to be prepared for the unexpected and to catch mistakes before they impact users

From <https://docs.microsoft.com/en-us/azure/architecture/framework/devops/overview>

High Avaliable - Network Architecture

Monday, July 12, 2021 1:43 PM

https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/dmz/nva-ha?tabs=cli

Resiliency - FMA - Failure mode analysis for Azure applications

Monday, July 12, 2021 1:44 PM

https://docs.microsoft.com/en-us/azure/architecture/resiliency/failure-mode-analysis

App Service

App Service app shuts down.

Detection. Possible causes:

- Expected shutdown
- An operator shuts down the application; for example, using the Azure portal.
- The app was unloaded because it was idle. (Only if the Always On setting is disabled.)
- Unexpected shutdown
- The app crashes.
- An App Service VM instance becomes unavailable.

Application_End logging will catch the app domain shutdown (soft process crash) and is the only way to catch the application domain shutdowns.

Recovery:

- If the shutdown was expected, use the application's shutdown event to shut down gracefully. For example, in ASP.NET, use the Application_End method.
- If the application was unloaded while idle, it is automatically restarted on the next request. However, you will incur the "cold start" cost.
- To prevent the application from being unloaded while idle, enable the Always On setting in the web app. See Configure web apps in Azure App Service.
- To prevent an operator from shutting down the app, set a resource lock with ReadOnly level. See Lock resources with Azure Resource Manager.
- If the app crashes or an App Service VM becomes unavailable, App Service automatically restarts the app.

Diagnostics. Application logs and web server logs. See Enable diagnostics logging for web apps in Azure App Service.

A particular user repeatedly makes bad requests or overloads the system.

Detection. Authenticate users and include user ID in application logs.

Recovery:

- Use Azure API Management to throttle requests from the user. See Advanced request throttling with Azure API Management
- Block the user.

Diagnostics. Log all authentication requests.

A bad update was deployed.

Detection. Monitor the application health through the Azure portal (see Monitor

Azure web app performance) or implement the health endpoint monitoring pattern.

Recovery:. Use multiple deployment slots and roll back to the last-known-good deployment. For more information, see Basic web application.

Azure Active Directory

OpenID Connect authentication fails.

Detection. Possible failure modes include:

1. Azure AD is not available, or cannot be reached due to a network problem. Redirection to the authentication endpoint fails, and the OpenID Connect middleware throws an exception.
2. Azure AD tenant does not exist. Redirection to the authentication endpoint returns an HTTP error code, and the OpenID Connect middleware throws an exception.
3. User cannot authenticate. No detection strategy is necessary; Azure AD handles login failures.

Recovery:

1. Catch unhandled exceptions from the middleware.
2. Handle AuthenticationFailed events.
3. Redirect the user to an error page.
4. User retries.

From <https://docs.microsoft.com/en-us/azure/architecture/resiliency/failure-mode-analysis>

Azure Cache for Redis

Reading from the cache fails.

Detection. Catch StackExchange.Redis.RedisConnectionException.

Recovery:

1. Retry on transient failures. Azure Cache for Redis supports built-in retry. For more information, see Azure Cache for Redis retry guidelines.
2. Treat nontransient failures as a cache miss, and fall back to the original data source.

Diagnostics. Use Azure Cache for Redis diagnostics.

Writing to the cache fails.

Detection. Catch StackExchange.Redis.RedisConnectionException.

Recovery:

1. Retry on transient failures. Azure Cache for Redis supports built-in retry. For more information, see Azure Cache for Redis retry guidelines.
2. If the error is nontransient, ignore it and let other transactions write to the cache later.

Diagnostics. Use Azure Cache for Redis diagnostics.

SQL Database

Cannot connect to the database in the primary region.

Detection. Connection fails.
Recovery:
Prerequisite: The database must be configured for active geo-replication. See SQL Database Active Geo-Replication.

- For queries, read from a secondary replica.
- For inserts and updates, manually fail over to a secondary replica. See Initiate a planned or unplanned failover for Azure SQL Database.

The replica uses a different connection string, so you will need to update the connection string in your application.

Client runs out of connections in the connection pool.

Detection. Catch `System.InvalidOperationException` errors.
Recovery:

- Retry the operation.
- As a mitigation plan, isolate the connection pools for each use case, so that one use case can't dominate all the connections.
- Increase the maximum connection pools.

Diagnostics. Application logs.

Database connection limit is reached.

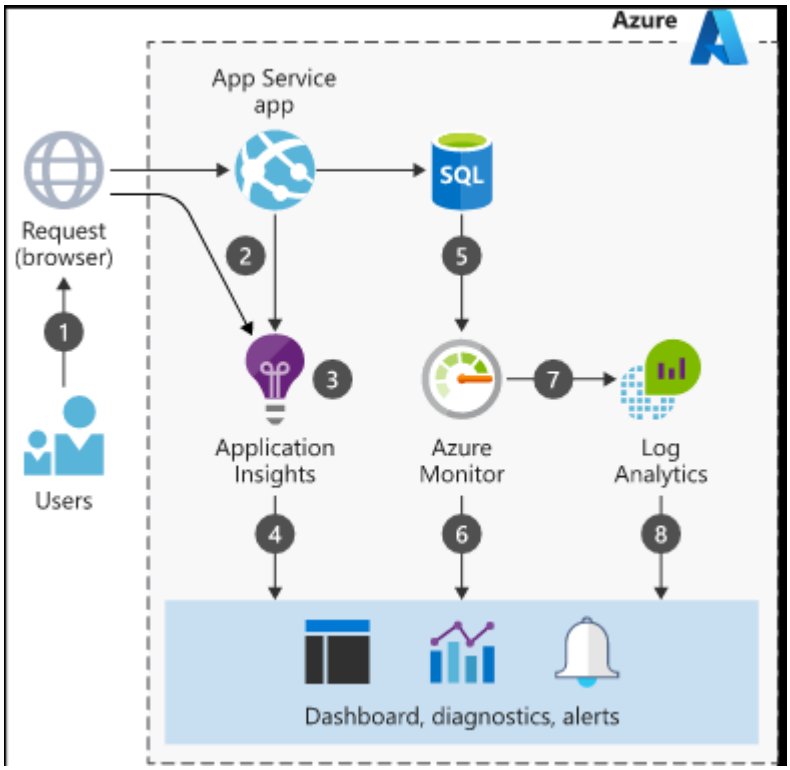
Detection. Azure SQL Database limits the number of concurrent workers, logins, and sessions. The limits depend on the service tier. For more information, see Azure SQL Database resource limits.
To detect these errors, catch `System.Data.SqlClient.SqlException` and check the value of `SqlException.Number` for the SQL error code. For a list of relevant error codes, see SQL error codes for SQL Database client applications: Database connection error and other issues.
Recovery. These errors are considered transient, so retrying may resolve the issue. If you consistently hit these errors, consider scaling the database.
Diagnostics. - The sys.event_log query returns successful database connections, connection failures, and deadlocks.

- Create an alert rule for failed connections.
- Enable SQL Database auditing and check for failed logins.

From <https://docs.microsoft.com/en-us/azure/architecture/resiliency/failure-mode-analysis>

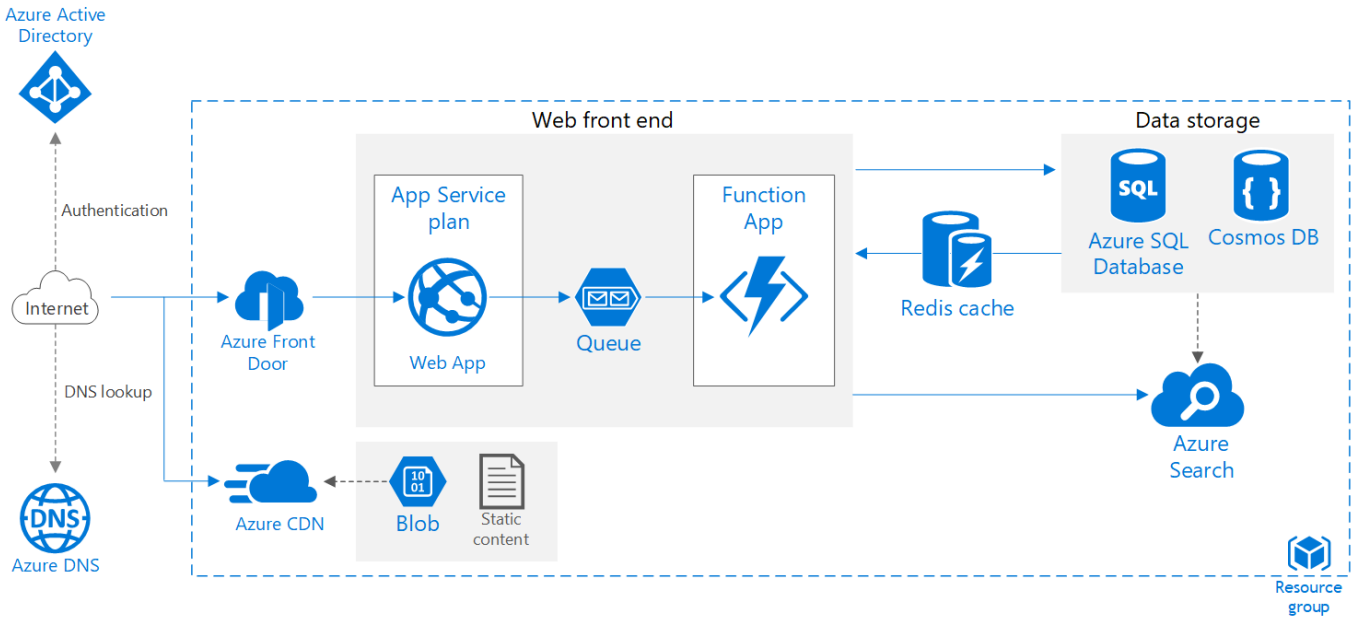
Webb APP Monitoring Arch

Thursday, July 15, 2021 3:41 PM



Scalable web application

Thursday, July 15, 2021 4:11 PM



Architecture

This architecture builds on the one shown in Basic web application. It includes the following components:

- Web app. A typical modern application might include both a website and one or more RESTful web APIs. A web API might be consumed by browser clients through AJAX, by native client applications, or by server-side applications. For considerations on designing web APIs, see API design guidance.
- Front Door. Front Door is a layer 7 load balancer. In this architecture, it routes HTTP requests to the web front end. Front Door also provides a web application firewall (WAF) that protects the application from common exploits and vulnerabilities.
- Function App. Use Function Apps to run background tasks. Functions are invoked by a trigger, such as a timer event or a message being placed on queue. For long-running stateful tasks, use Durable Functions.
- Queue. In the architecture shown here, the application queues background tasks by putting a message onto an Azure Queue storage queue. The message triggers a function app. Alternatively, you can use Service Bus queues. For a comparison, see Azure Queues and Service Bus queues - compared and contrasted.
- Cache. Store semi-static data in Azure Cache for Redis.
- CDN. Use Azure Content Delivery Network (CDN) to cache publicly available content for lower latency and faster delivery of content.
- Data storage. Use Azure SQL Database for relational data. For non-relational data, consider Cosmos DB.
- Azure Cognitive Search. Use Azure Cognitive Search to add search functionality such as search suggestions, fuzzy search, and language-specific search. Azure Search is typically used in conjunction with another data store, especially if the primary data store requires strict consistency. In this approach, store authoritative data in the other data store and the search index in Azure Search. Azure Search can also be used to consolidate a single search index from multiple data stores.
- Azure DNS. Azure DNS is a hosting service for DNS domains, providing name resolution using Microsoft Azure infrastructure. By hosting your domains in Azure, you can manage your DNS records using the same credentials, APIs, tools, and billing as your other Azure services.

From <https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/app-service-web-app/scalable-web-app>

API Management - Availability and scalability

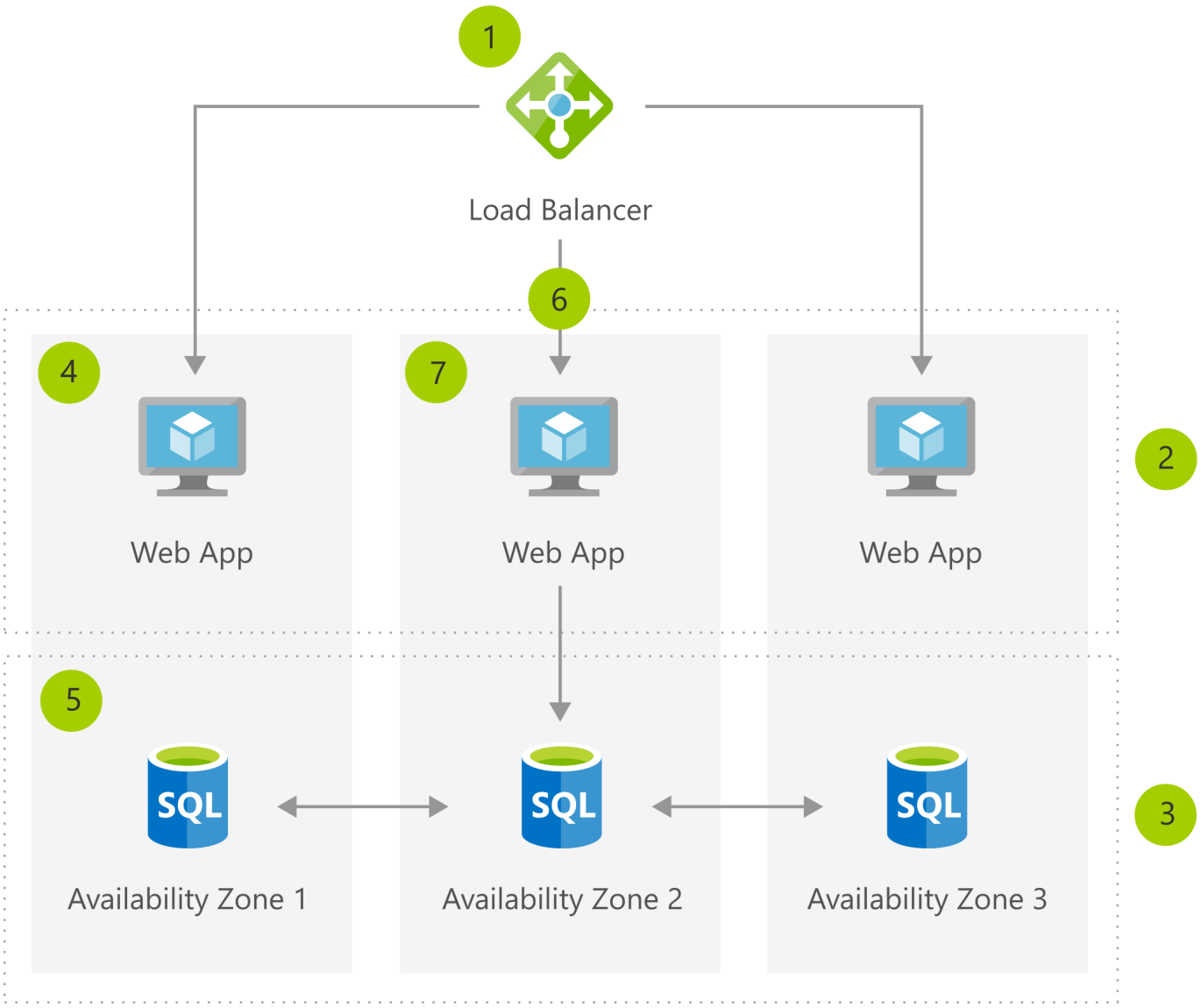
Thursday, July 15, 2021 4:27 PM

- Azure API Management can be scaled out by choosing a pricing tier and then adding units.
- Scaling also happen automatically with auto scaling.
- Deploying across multiple regions will enable fail over options and can be done in the Premium tier.
- Consider Integrating with Azure Application Insights, which also surfaces metrics through Azure Monitor for monitoring.

From <https://docs.microsoft.com/en-us/azure/architecture/example-scenario/apps/apim-api-scenario>

Build high availability into your BCDR strategy

Thursday, July 15, 2021 4:29 PM



1. Create zone-redundant Load Balancer.
2. Create front-end subnet.
3. Create DB subnet.
4. Create VMs in three Availability Zones.
5. Configure zone-redundant SQL DB.
6. Add VMs to the load balancer's back-end pool.
7. Deploy your application on VMs for redundancy and high availability.

From <https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/build-high-availability-into-your-bcdr-strategy>