DES424 – Final Exam Paphana Yiwsiw 6222780379

*** 06 – Cloud Network & Security ***

Traditional Approach - usually performed by dedicated hardware, such as routers, delivery controllers. / SDN: Software-based, communicates via API, more freedom, manage resources virtually on the control plane.
Virtual Network (VN) - Networking system that emulates a physical network, combining hardware and software to a single administrative unit. SDN can create and control VN or control hardware using software.
Components: *vSwitch*: software on host server, allow VN configuration, control and direct communication between physical network and virtual parts. / *VN adapter*: a gateway between networks / *Physical network*: host of VN / *VMs*: machines to connect / *Servers*: part of infrastructure / *Firewalls and security*: monitor and prevent security threats in VN / *Virtual NIC*: connect to vSwitch or bind to physical NIC [Binding mode: Bridge (direct), NAT (Host IP shared), VN (VLAN, segment)]
Network Address Translation: NAT -Remapping single IP to another by modifying network address info in IP headers during transiting across routing devices. Allow private IP devices to communicate.
3 Classes of VN – [VPN/VLAN/VXLAN] *VPN*: Virtual Private Network Internet as a transport, connect 2 networks or remote client to central network, create secure and encrypted tunnel between endpoints, obscuring traffic and ensure the privacy, all services on both sides are presented as local services but hidden in between. *VLAN*: Virtual Local Area Network: subdivides LAN into separate LAN, each VLANs act as a separate LAN, allow more security, monitoring, and management. *VXLAN*: Virtual Extensible Local Area Network. Strech layer 2 conn. over layer 3 network, encapsulating ethernet frames in VXLAN packet which includes IP addresses for scalability problem in more extensible manner.
VN Benefits - *Management:* segmenting and joining virtually provide better access management. *Security:* traffic segmentation and access restriction. *Simplification*: replace rigid network and hardcoded routes with simplified optimized path. *IP mobility:* VMs can move to different hosts across the network while maintain the same IP, helps when recovering from disasters and balancing load.
VN on Cloud - *Resource group*: based on Azure account to control resources. *Default VN*: default network assigned when create VM. *Network Security Group*: control security for devices and services in the group. [*Activate rule or ACL*: Allow/Deny network traffic to VM in VN. NSGs can be associated with subnets or VM instances inside subnet.] *Azure VPN Gateway:* VPN allow connecting on-premises with cloud network. *Azure NAT Gateway:* Provides outbound internet connection for subnets in the VN. NAT provides source NAT for that subnet after NAT associated. Specifies which static IP VM used when creating outbound
IAM: Identity and Access Management - Mechanism that enforces policies to control and track user identities and access for IT resources. *Authentication*: username/password is the most common form of auth credentials. *Authorization*: defines the correct granularity for access controls and supervise relationships between identities, access control rights, and resource availability. *User Management:* responsible for creating new user identity and access group, resetting passwords, defining password policies, and managing privileges. *Credentials Management:* establishes identities and access control rules for user accounts, mitigates threats of insufficient authorization. *AAA*: Authentication(valid?), Authorization(permit?), Accounting (used?)
Authentication Types - API/Token or App
- *Internal:* Username/password, maintain internally. Password must be encrypted and use secure connection. <u>*Password hashing:*</u> no plaintext password, stored hashed, compare hash code with the hashed for authentication. (pwd+salt)→ hash → stored salt and hash
- *External:* SSO(LDAP/AD/Cookie), OAuth, Open ID (access only)

-- *SSO: Single Sign-On:* Enable users to authenticate with app using one set of credentials. Auth servers maintain passwords for all app, pass an authentication token to external apps and services, stored in cookie. Advantages: no repeated password, policy enforcement, MFA, internal credential management › external storage.
-- *AD: Active Directory*: Allow IT dept. to manage and store info about users and devices within org network, provided centralized user mgmt. Advantages: easier to organize users, manage file access quicker, assign access right appropriately, maintain an overview of resources.
--- Azure AD: a cloud based IAM services, helps user access external resources and internal resources for your own organization.
--- MFA: layered approach to securing data and app, a system requires a user to present combinations of credentials to verify login identity of users. (Username + password then OTP/Google Authenticator)
-- *OAuth 2.0*: Open Authorization, a way for users to grant websites or app to access to their information on others without giving them passwords. No password sending to other apps. Require redirect URI to redirect back to app after authentication finished. *OAuth refresh token:* string that OAuth client used to get a new access token without user interaction. Must not allow the client to gain any access beyond the scope of original grant.
-- *OpenID:* SSO for consumer apps, login only, SAML 2.0: open standard for authorization and authentication, SSO for enterprise apps. XML form.
User Management Structure - RBAC (Role) or ABAC (Attribute-based)
- *User Data:* Masked password: type password and confirm again.
- *Roles*: support multiple roles for a user, Role, Role Access, Operation: show list of operation and stored in ROLE_ACCESS - *Operation*: Support hierarchy and order of operation. - *User mobile*: store registration ID for sending notification, app version, device type, device version, last access date. - *Activity log*: logging activities both web and API. - *Config*: store key/value of configuration that can be changed e.g., mail server info, password policy, image path, app version

Simplified Secure API with Token - Mapping token with user account
- Control access of the token: from host IP, referrer domain, limit usage period, bypass mode for test. Use hash for generate, keep usage log.

*** 07 – Microservices ***

Traditional App Architecture: Monolithic: built as single and indivisible unit. Comprises of client-side UI, server-side app, and a database. Unified and functions are managed and served in one place. Drawbacks: tightly coupled between components, less usability, large code base, tough for dev and QA to understand, less scalable, doesn't follow *SRP: Single responsibility people*, more deployment and restart times, new technology barriers (app need to be rewritten)
Monolithic to Microservice - A microservices architecture breaks down into a collection of smaller independent units. All services have their own logic and the database as well as perform the specific functions. The app can access microservices through direct call
- Advantages: Lower cost, more efficiency, increased scalability, easier maintenance and update, fault tolerance, independent deployment.
- Disadvantages: higher complexity, increase traffic, increased dev time, dev complexity, difficult global testing and debugging.
Microservice Architecture
- *Clients*: web browser, mobile, container client. Different consume functionality from multiple microservices and require frequent updates.
- *Identity Providers*: permit server-to-server and user-driven access.
- *JWT*: creating data with signature and encryption, payload contains JSON of claims, tokens are signed either private secret or RSA key.
- *API Gateway:* single app entry point, routing and translate protocol.
- *Messaging formats:* Sync or Async messages

-- *Synchronous Message:* REST, clients wait for responses, relies on stateless, client-server, and HTTP protocol.
-- Asynchronous Message: client doesn't wait for responses, use AMQP, STOMP, MQTT. Used in nature of messages is defined and interoperate between implementations. Popular message brokers are RabbitMQ and Apache Kafka. Model: One-to-one or Publish-and-Subscribe
- *Data Handling/Databases:* Each microservice owns a private database to capture their data and implement business functionality. Maintain *ACID properties*: Atomicity, Consistency, Isolation, and Durability.

When using ***single database cluster***, leveraging the schemas or tables. RULE OF THUMB: writing is allowed from microservices that own the data; reading may be allowed to any services needed.
When using ***multiple databases***, duplicate the data e.g., AWS DMS Writing is only allowed by microservice associated with the database; reading is allowed from any database where the data is duplicated.
- *Service Discovery:* Manage deployment and load distribution
- Service provider: originates service instances over the network
- Service registry: DB that stores the available location of instances
- Service consumer: retrieves the location of a service from registry.

-- 2 Major discovery patterns: Client-side or Server-side
--- Client-side; searches the registry to locate a service provider, selects available instance with load balancing, make a request.
--- Server-side discovery pattern; load balancer searches the service registry and forwards request when once applicable service found.
- *Management:* allow business users to configure service during run-time such Docker / Kubernetes / Openshift / Istio
- *Static Content:* deploy static content to a cloud storage service that can deliver them directly to client using CDN
- *Logging:* Send all generated logs across the hosts to a centralized external place while microservices are running on multiple hosts.

Microservice Deployment server, VM, container, or cloud.
- *Single Machine, Multiple Process:* Basic form, group of processes with load balancing. Benefits: Lightweight, convenience, easy fix, fixed billing Limits: No scalability, single point of failure. App pack such as exe, JAR, WAR.[Source Code → CI/CD → Prod. / CI build artifacts, CD retrieved.]
- *Multiple Machine, Multiple Process:* Scale up (upgrade server) or scale out (add more server) when the app's capacity is exceeded.
- *Containers:* running microservices directly as processes is efficient. The server must be maintained with dependencies and tools, runaway process can consume all memory/CPU, deploying and monitoring on many servers can be troublesome. *Containers: Packages that contain what program needed, provide enough virtualization to run in isolation. Container image* is a self-contained unit that can run on any server without installing any dependencies or tools. Benefits: Isolation, Concurrency, less overhead, No-install deployment, Resource control
- *Serverless Container:* CaaS e.g., AWS Fargate and Azure Container run containerized app without dealing with servers. ECS with Fargate allows us to run containers w/o renting servers, maintained by cloud providers.
- *Orchestrators:* Platforms specialized in distributing container workloads over a group of servers. Most well-known: Kubernetes [Custom deployment scripts, codify the desired state with a manifest file and let the cluster take care of the rest.]
- Serverless Functions: Use the cloud to build and run code on demand. AWS Lambda and Azure functions handle all infrastructure details required for scalable and highly available services.

*** 08 – Cloud Native & Docker ***

Cloud App Type: Monolithic, Cloud-Enabled, Cloud-Based, Cloud-Native
Cloud-Enabled App: Built traditionally, migrated to cloud, designed in monolithic, depended on local resources H/W. Refactored to virtual resources, remained same underlying architecture. No advantage of shared services or resource pool, issue with scalability
Cloud-Based App: Middle ground b/w cloud-native and cloud-enabled. Leverage some capabilities e.g., higher availability and scalability, but don't want to redesign whole app to use cloud services. Adopt IaaS, PaaS, SaaS. No need to maintain a server or worry about backup.
Cloud-Native App: Architected to run in a public cloud using cloud tech, allow more accessibility and scalability. Comprised of CI, orchestrators, and container engines. Core is microservices.
DOCKER: Standardized, executable components that combine app source code with OS and dependencies required to run in any environment. Like VM, but lightweight, smaller, faster, and easier to manage. *Architecture:* client-server; docker client talks to daemon (building, running, and distributing), via REST API over UNIX sockets or network interface. Daemon will pull images from registry and run those as containers. (Users interact with client, daemon execute command)
- Communicate with client using CLI; <u>build, push, pull, run, start, stop</u>.

Docker CLI command:
Run a new container:
```
docker run IMAGE [--name (container name), -p
(host:container), -P (map all port), -d (detach), --
hostname (assign to host name), -v (host dir:target
dir), -it --entrypoint (EXECUTABLE IMAGE)]
```
- Build image from Dockerfile:

```
docker build -t IMAGE DIRECTORY
```
List: `docker ps [-a (all)]` *Delete:* `docker rm [-f (force)]`
Start: `docker start` *Stop:* `docker stop`
-Start shell inside: `docker exec -it CONTAINER EXECUTABLE`
- Run command from container: `docker exec CONTAINER COMMAND`
- Download/Upload image: `docker pull/push IMAGE[:TAG]`

Docker registry: store docker images, docker hub, docker pull request is made to pull image from docker registry, push command to store.
Docker image: execute code in docker container, image = set of instructions to build docker container. Image is equivalent of a snapshot in VM, more lightweight, act as starting point. - *Where to store data:* let docker mange the storage with own internal volume management, or create directory on host system and mount to container from inside
- *Docker image* is made up from bundled files, with all essentials and dependencies, required to configure a fully container environment.

-- Interactive method: saving the result state as new image.
--- ***Dockerfile:*** provides the specification for creating a docker image.
--- Leftmost word = instruction, right is arguments of those instructions.
<u>▤ **Basic in Dockerfile: *`FROM, LABEL, RUN, WORKDIR, EXPOSE, CMD`***</u>
- **`FROM`**: specify base image to be pulled from docker hub
- **`RUN`**: execute command during image build process
- **`ENV`**: set environment variables, both build and running time.
- **`COPY`**: copy local files and directories to the image
- **`EXPOSE`**: specify port to expose
- **`ADD`**: can COPY from URL; not recommended, use curl, or using RUN.
- **`WORKDIR`**: set current working directory for this Dockerfile.
- **`VOLUME`**: create or mount the volume of the container
- **`USER`**: set username and UID when running container
- **`LABEL`**: specify metadata of docker image
- **`ARG`**: set build time variables, with key and value.
- **`CMD`**: execute command in running container, only last CMD is applied.
- **`ENTRYPOINT`**: execute when container starts. Default: /bin/sh -c

Good practice of Dockerfile: lightweight base, install only what to use. Use .dockerignore to exclude files/directories.
- *Docker container:* instance of image, managed through docker API. Lightweight and independent. Data will be scrapped when removed.
- *Docker daemon:* fulfill container actions, run background, manage network, storage, volume, container, images. Build when requested.
- *Docker networking:* allow containers to communicate with others, provide services outside via host network, support multiple networks.

-- None: disable network systems, no connection with other containers
-- Bridge: default driver, used when containers connect w/ same host. create private network to host, container on network can communicate.
-- Host: when user doesn't require isolation between container and host. Use host's IP and TCP port to expose service running inside container
-- Overlay: allow communications. between swarm services, when run on diff hosts. Simplified version in multi-host networking.
-- MACVLAN: make container look like physical driver, assigned MAC address and route traffic b/w containers through. Connect container interface directly to host's, addressed with routable IP addresses that on subnet of external network, map subnetwork to another via VLAN.
- *Docker Storage:* enable storage admin to configure and support app data storage within container deployments. Used storage drivers to manage the content and writable container layer. By default, containers don't write data permanently to any location, storage must be config if want to store data permanently. If stored outside, usable even removed.

-- Layers: docker built 1 READ-ONLY layer for each instr. in the Dockerfile.
-- When command run is executed, docker builds READ-WRITE layers
-- New file can be created on the container
-- <u>*Copy-on-Write Mechanism:*</u> Modify existing files in the image layers, local copy is created on the container layer, change live on container. Files and modifications are destroyed when container is destroyed. Use volume mapping techniques to persist the data.
--- Types of storage: Permanent[Volume/Bind Mount], Temp[tmpfs]
---- Docker Volumes: area of isolated storage on host contains running docker engine. Same volume can use for the next or different container.
---- Bind Mount: May be stored anywhere on the host system, even important directories. Non-docker processes can modify any time.
---- Tmpfs: Data written directly on host's memory, deleted upon stop. Useful when involved sensitive data or don't want it to be permanent.

Lifecycle: Build (build)→ Push (push)→ Run (run)→Upgrade (stop)
Docker Swarm: Container orchestration tool, allow user to manage containers deployed across multiple hosts. benefit: high availability.
Example of Dockerfile (Web with nginx)

```
LABEL maintainer="contact@dev.com"
FROM ubuntu:18.04
RUN apt-get -y update && apt-get -y install nginx
COPY files/default /etc/nginx/sites-available/default
COPY files/index.html /usr/share/nginx/html/index.html
EXPOSE 80
CMD ["/usr/sbin/nginx","-g","daemon off;"]
```

*** 09 – Docker Compose ***

- A tool in defining and sharing multi-container app, define services in YAML file while tear the composed container in a single command.
- 3 steps: **Define** environment with Dockerfile, **define** services in app, **run** docker compose up and docker compose starts and run the app.
YAML - Data serialization language for writing configuration files, human-readable & easy to understand. Python-style indentation for nesting, Tab is not allowed, whitespaces are used instead. No format symbols e.g., braces, square, brackets, closing tag, quotation marks. For comment use #. Node anchors: mark with &, reference with *
YAML for docker compose - Usually contain at least 1 service, optional for volumes and network. Services refer to containers' configuration. Service can be image from docker hub or build from Dockerfile. Ex. web app with front/back/database, split into 3 images and define as services.
- *Volumes* define physical disk shared between host and container
-- 2 types: named and host ones. *Named volume:* docker managed, auto mounted in self-generated directory. *Host volume:* specify existing folder in host. Host at service, named volume at global/outer level
- *Networks* define communication between containers and between container and host. Service can communicate with another on the same network by referencing by container name and port.
- *Environment variables:* Define static env variables and dynamic variables with ${} notation. Methods: [.env file in the same directory: key=value/ OS export before docker-compose up: export key=value / Inline command: KEY=value docker-compose-up]
<u>Example of docker-compose.yml</u>

```
services:
  db:                        # database service
    image: mysql:5.7         # specify image
    volumes:
     - db_data:/var/lib/mysql
    restart: always          # always restart
    environment:             # set env var
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
  wordpress:                 # web app service
    depends_on:              # set service dependency
     - db                    # depends on service db
    image: wordpress:latest   # specify image
    ports:
     - "8080:80"             # Map port with host port
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
volumes:  # volume that is visible to all services
  db_data:
```

Lifecycle management
1. First time running: `docker-compose up`
2. Start services (not the first time) `docker-compose start`
3. Different file name than default docker-compose.yml (-f for alternate file name) `docker-compose -f custom-compose-file.yml start`
4. Run in background (-d for detach mode) `docker-compose up -d`
5. Shutdown `docker-compose stop`
6. Deletion (delete everything) `docker-compose down`

*** 10 Software Testing & CI/CD ***

Failure: deviation of observed behavior from specified behavior
Error state: System in state such further processing can lead to failure
Fault: mechanical or algorithmic cause of an error a.k.a. bug
Validation: checking activity of deviations b/w observed and specs.
Example of faults and errors: *Interface faults*: client/server mismatch, requirement/implementation mismatch. *Algorithmic faults:* missing initialization, incorrect branching, null test missing. *Mechanical faults:* operating temperature outside equipment specifications *Errors:* null reference, concurrency, exceptions. *Relative cost of bugs:* cost to fix a bug increase exponentially, "found later, cost more".
How to deal with faults: *Avoidance:* Method to Reduce complexity. Use configuration management to prevent inconsistency. Verification to prevent algorithmic faults. Reviews. *Detection:* Testing: provoke failure in planned way. Debugging: Find and remove cause of faults of observable failure. Monitoring: Deliver information about the state, used during debugging. Tolerance: Exception handling, modular redundancy Developing effective test. You must have detailed understanding of the system, app and solution domain knowledge, testing techniques, skill to apply these techniques. Testing is done by independent testers.
Testing Activities: *Unit test:* Object Design. *Integration test:* System Design. *System test:* Requirements analysis. *Acceptance test:* Client.
Testing Types: Unit, Integration, System, Acceptance.
- ***<u>Unit testing:</u>*** done to individual components, done by developers. Goal: Confirm it coded correctly and carried out intended functionality
-- Black-box testing: I/O behavior; predict the output for any given input. Goal: reduce test cases number by equivalence partitioning: divide input condition equally, then choose test case for each.
-- White-box testing: Path testing: all paths in modules must be used at least once. Loop testing: start and end conditions works properly Condition testing: execute, all possible outcomes come at least once.
-- Test oracle: mechanism to tell pass or fail test
- Unit test heuristic: Create when object design is finished, black for functional, white for dynamic. Develop test case: find effective number of test cases. Cross-check test cases to eliminate duplicates. Desk check your source code. Create test harness: test drivers and stubs are needed for integration testing. Describe test oracle, usually first passed one. Compare the results of test with test oracle, automate if possible.
-- JUnit: Java framework for writing and running unit tests. Written with "test first" and pattern-based development.
- ***<u>Integration testing:</u>*** done to group of subsystems, eventually entire system. Done by developers. Goal: Test interface among subsystems. Determines the order in which subsystems are selected for testing and integration because unit tests run only in isolation and many failures caused by interactions of subsystems. Without this, system testing will be very time-consuming. Failures that are not discovered during this time, it will be discovered after deployment which is costly to fix.
-- *Driver:* component called tested unit, control test cases, main prog.
-- *Stubs:* tested unit dependent component, partial implementation, return fake values. Example: Login can be driver (dummy login page), Admin page is stub, acting as called program from admin.
-- ***Bottom-up testing strategy:*** Subsystem in the lowest later of call hierarchy are tested individually. Next subsystems are tested that call the previously tested subsystems. Drivers are needed, No stubs. Suitable for object-oriented or real-time system
-- ***Top-down testing strategy:*** Test top layer first, then combine all subsystems that are called by the tested system and test the resulting collection of those subsystems. Do until all subsystems are tested. Many stubs are needed to do the testing, no drivers. Test cases can be defined based on the functional requirements.
-- ***Continuous Testing:*** Continuous build: build/test/integrate from day one. Always runnable. Required integrated tool support: continuous build server, high coverage automated test, refactoring supported tool, software configuration management, issue tracking.
--- Integration Testing Steps: Select component to test, unit test in component first. → Combine selected component, preliminary fix-up, make drivers and stubs. → Test functional requirements: define test cases that go to all use cases. → Test subsystem decompositions: define test case with all dependencies. → Test nonfunctional components: performance tests → Keep records of test cases and activities. → Repeat until the system is fully tested. Goal of integration testing: identify failures with component configuration.
- ***System testing:*** Testing done to entire system. Done by developers. Goal: Test system meets functional and nonfunctional requirements Functional Testing: validates functional requirements. Test cases designed from requirement analysis docs or user manual, centered around use cases (requirements/key functions). Treated as black box, unit test can be reused along with newly-develop test cases
-- *Performance Testing:* validates non-functional requirements. Test how system behaves when overloaded; identify bottlenecks etc. Try unusual execution order, check system's response to large volumes of data, amount of time spent in use cases. Type of performance testing:
• Stress test: limit. • Volume test: handle large data. • Config test: test software and hardware config. • Compatibility test: test backward compatibility. • Timing test: response time and function performing time. • Security test: violating security requirements. • Environmental test: test tolerances in physical environment. • Quality test: test reliability, maintainability, availability. • Recovery test: loss of data response from the system. • Human factor test: test with end users.
--- Web app testing: Load/Stress/Volume/Lifecycle/Optimization/Soak
--- Tools: loader.io, blazemeter.com, JMeter (Apache, pure Java app)
- ***Acceptance testing:*** Evaluate the system from developers, Done by client. May involve executing typical transaction on site in a trial basis Goal: Demo that the system meets the requirements and READY TO USE.
-- ***Alpha test:*** by client, use dev's environment in controlled settings.
-- ***Beta test:*** use client env w/o dev supervision, work in target env.
4 Testing Steps: 1. Select what to test: Analysis/Design/Implementation 2. Decide how to test: Review or inspect code/Proofs/Black or white /Testing strategy. 3. Develop test cases: *test case is a set of test data or situations that will be used in that unit for testing or about the attribute being measured.* 4. Create test oracle: contain the predicted results for a set of test cases, written down before actual testing takes place.
CI/CD - A method that frequently deliver apps to users with automation in stages in app development. A solution to the integration of new code.
- *CI:* practices that involves dev making small changes and check codes.
- *CD (Delivery):* automated delivery of completed code to testing env.
- *CD (Deployment):* next step of CD, every passed-test change automatically placed in prod, resulting in many productions deployments
- *Pipeline:* code, commit, build, test, stage deploy, production deploys
- CI/CD is a set of development practices to deliver code changes rapidly and reliably. Focused on SDLC, highlight tools, emphasize automation.
- ***DevOps*** is a collection of ideas, practices, processes, and technologies that allow development and operations team work together to streamline product development. More focused on culture, highlight roles, emphasize responsiveness.
CI/CD Tools: *Jenkins:* CI/CD automation software DevOps tool in Java, implement workflow/pipelines. *Robot framework:* acceptance testing automation framework, test-driven development, uses keyword-driven testing approaches. Library: Selenium Library.

*** 11 – Scaling Services ***

Dapr is a system to support cloud native and serverless computing. Portable and event-driven runtime. Easy for dev to build resilient, stateless, stateful cloud microservices and embraces language diversity and developer frameworks. *Sidecar pattern:* About separating cross-cutting operations from microservice to reduce internal complexity. Dapr exposes HTTP and gRPC API as a sidecar pattern, either container or process, not requiring code to include any dapr runtime code.
Azure App Insights - When building microservices app with many distributed services that communicate with each other and use different infrastructure, then it is required the mechanism to observe and trace from end to end. - App Insights is a tool to monitor azure container apps under the same container app environment and collect service telemetry and usage and engagement via integrated analytics tools.
- Telemetry: data collected to observe app, 3 categories.
-- Distributed tracing: insights into traffic between services in distributed transactions. Ex. Front-end talks with back-end to get data.
-- Metrics: insights into performance of a services and resource utilization. Ex. CPU and memory utilization of background processor, help us to understand when we need to increase the number of replicas.
-- Logging: insight into code execution and occurred errors.
- Investigate tab in app insight
-- Services connectivity: for distributed tracing via app map
-- Live metrics: near real-time status of distributed app graph.
-- Transaction search: find and explore telemetry items e.g., exceptions, requests, events.
-- Failure: view frequency of failures across different operations
-- Performance: display performance details, identify longest duration, diagnose problems.
Azure CLI Command

```
az login
az account show
az extension add -n containerapp -upgrade
az extension add -n app-insights
az provider register --namespace Microsoft.App
az provider register --namespace Microsoft.OperationalInsights
az group create --name ¬myrg --location "eastus"
az acr create --resource-group myrg --name ACRName --sku Basic
--admin-enabled true
az acr build --registry ACRName --image "IMAGEName"
--file "PathtoDockerfile" .
az containerapp env create --name ContainerAppName
--resource-group myrg --location "eastus" --dapr-instrumentation-
key InstrumentKeyfromAppInsight
az containerapp update --name "ContainerAppName"
--resource-group myrg --set-env-vars ENV_VAR=VALUE
az containerapp env dapr-component set --name "ContainerAppName"
--resource-group myrg –dapr-component-name statestore
--yaml "File.yaml"
az containerapp create
--name ContainerAppName             # container app name
--resource-group myrg               # resource group name
--environment ContainerAppName      # container app environment
--image "ACRName.azurecr.io/ImageName" # container image in ACR
--registry-server "ACRName.azurecr.io" # registry server
--target-port 80     # target port (listen for incoming request)
--ingress 'external' # external = from public, assigned FQDN
                     # internal = w/in Azure container Environment
--min-replicas 1 --max-replicas 2
--cpu 0.25 --memory 0.5Gi --query configuration.ingress.fqdn
# If you use dapr
--enable-dapr --dapr-app-id DAPRAppID --dapr-app-port 80
az cosmosdb check-name-exists --name CosmosDBName
az cosmosdb create --name CosmosDBName --resource-group myrg
az cosmosdb sql database create --account-name CosmosDBName
--resource-group myrg  --name DatabaseName
az cosmosdb sql container create --account-name CosmosDBName
--resource-group myrg --database-name DatabaseName
--name SQLAPIContainerName --partition-key-path "/id"
--throughput 400
az cosmosdb keys list --name CosmosDBName --resource-group myrg
Add secret to container app
az containerapp secret set --name ContainerAppName
--resource-group myrg --secret "KEY=VALUE"
az containerapp update --name ContainerAppName
--resource-group myrg  --image "ACRName.azurecr.io/ImageName"
--revision-suffix RevisionSuffix --cpu 0.25 --memory 0.5Gi
--min-replicas 1 --max-replicas 2 --set-env-vars
"AppInsight__InstumentationKey=secretref:appinsight-key"
```

Robot Framework

```
*** Settings ***                    # Import Library
Library  Selenium2Library
*** Variables ***                   # Define Variables
${BROWSER}  chrome
${URL}  http://localhost:3000
*** Test Cases ***                  # Testing Steps
1. Open Browser
  Open Browser  ${URL}  ${BROWSER}
  Maximize Browser Window
  Set Selenium Speed  0.5
2. Go to Log In Page
  Click Element  id=nav-user-icon
  ${PAGE_TITLE}  Get Text  xpath=//h1
  Log To Console  ${PAGE_TITLE}
  Should Contain  ${PAGE_TITLE}  Log In
3. Enter Log In Account
  Input Text  id=login-username-input  ${USERNAME}
  Input Password  id=login-password-input  ${PASSWORD}
  Click Button  id=login-signin-btn
```