|  |  |
| --- | --- |
| Kubernetes Fundamentals | |
| K8s resources | Resources: objects of a certain type in the K8s API  - Interact w the cluster by creating and modifying these objects |
| Pods: most basic resource in K8s  - Represents a group of 1 or more containers  - Pods resource: abstract grouping of ≥ 1 pod objects  - Init container: to run a task before a Pod’s main container starts up |
| Resource Types: objects/resources have a resource type  - Object’s resource type determines K8s functionality controlled by that obj  - *kubectl api-resource* : list all available resource types in cluster  - *kubectl explain <resource> :* get documentation about a resource  - Can define own custom resource types using a CustomResourceDefinition |
| Kubernetes obj: piece of data that represents the state of the cluster  - When you create/modify an obj, K8s works constantly to implement the desired state expressed in the obj data  - *kubectl apply -f <yaml file>* : apply file configs to cluster  - *kubectl get <resource>* |
| Managing Pods | 1) ReplicaSet: ensures a given num of replica Pods are running at any given time  - Creates Pods from a template, or deletes them to meet the desired replica count |
| 2) Deployment: similar to ReplicaSet but provides a way to declaratively update Pods and ReplicaSets. – Great for scaling stateless apps  - Uses a RollingUpdate deployment strategy by default for zero-downtime updates to workloads  - Declarative method: *kubectl apply -f <deployment yaml file>*  - Imperative command: *kubectl create deploy <name> --image=<image> --replicas=<replicas>* |
| 3) StatefulSet: similar to Deployment, but for stateful Pods  - Pods maintain order and a sticky identity (same Pod name, network hostname), even if they are re-created on a diff Node  - Requires manual creation of a headless Service alongside the StatefulSet (headless Service is Service type w clusterIP: None) |
| 4) DaemonSet: runs a replica Pod on each Node  - Dynamically create new replicas for new Nodes as they join the cluster  - Can use filtering criteria to only run on some Nodes |
| 5) Job: runs a containerized task to completion  - Auto retries if the containerized task fails |
| 6) CronJob: runs jobs repeatedly accordingly to a schedule |
| K8s Architect-ure | Control Plane: set of components that manage worker Nodes and Pods  Worker Nodes: Host Pods and run container workloads |
| In Control Plane: API Server, etcd, Scheduler, Controller Manager, Cloud Controller Manager  1) API Server: center of control plane.  – Other components (and users) use it to communicate and interact w the cluster  2) etcd: reliable object storage  - Used by API Server to store data about the state of the cluster  - Storage is distributed and can have multiple nodes for reliability  3) Scheduler: watchers for new Pods that have not yet been assigned to a worker Node  - Selects an appropriate Node to run each new Pod on (process known as scheduling)  4) Controller Manager: combines multiple controllers into a single process  - Each controller provides diff functionality to the cluster  - E.g. Job controller watches for new Jobs and creates Pods to execute the Job workload  5) Cloud Controller Manager: runs controllers that interact w cloud provider APIs and features  - Only runs controllers needed for the cloud provider you are using  - Non-cloud clusters don’t have a cloud controller manager |
| In Worker Node: kube-proxy, kubelet, Container Runtime  1) kube-proxy: acts as a network proxy to implement parts of the K8s Services feature  - Maintains network rules on Nodes to route traffic to Pods  2) kubelet: K8s agent running on each Node  - Ensures the containers specified by each Pod on the Node are running  - Works w the container runtime to manage containers  3) Container Runtime: software responsible for running containers, such as containerd or CRI-O  - Any runtime that implements the K8s CRI (Container Runtime Interface) standard will work  - kubelet no longer uses dockershim to support Docker as a container runtime, since Docker don’t implement the CRI |
| K8s API | HTTP API that allows users, cluster components, and external components to communicate  - Allows you to query and manipulate the state of K8s objects  - k8s API is a RESTful or REST API: can retrieve a representation of the state of a resource/object  OR can send a representation to the state to create/update objects  - Can use formats like YAML or JSON, but objects are ultimately considered REST or RESTful objects |
| Interact w the API via HTTP verbs like GET, UPDATE, DELETE  - API persists the corresponding object data in etcd |
| Containers | Container: self-contained package containing code and all its dependencies  - Can run w specialized isolation from other processes on a host  - Isolation provided using Linux control groups  - Can have 1 or more containers in a Pod |
| Dockerfile: text file containing instructions/commands used to build a container image |
| Scheduling | Process of assigning a new Pod to a Node  Scheduler takes a variety of factors into account when choosing a Node  - Resource requirements: assign Pods only to Nodes w sufficient resources  - Pod affinity (selector) and anti-affinity: Deep customization around scheduling  - Taints/tolerations: Taints prevent Pods from scheduling on Nodes by default (e.g. control Node) |
| *nodeSelector:*  *nodeclass: <label>* |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Container Orchestration | | | |
| Intro | | Container Orchestration: automation of the work required to run & managed containerized workloads  Manual workload: performs tasks like deploying, running, restarting containers manually  Orchestration: tasks are automated | |
| K8s is a container orchestration tool  - Deploying Containers: K8s scheduler auto assigns Pod to Node. kubelet handles spinning up containers on that Node  - Restarting Broken Container: kubelet auto restarts containers that are unhealthy. K8s liveness probes let you customize how kubelet detects container’s health status | |
| Container Runtime | | | Software responsible for actually running containers  kubelet on each worker node communicates w the container runtime to manage containers  Container runtime is not part of K8s, but installed separately  - Multiple options for container runtimes that works w K8s |
| Container Runtime Interface (CRI): standard protocol for communication btw kubelet and container runtime. kubelet -> CRI -> container runtime |
| E.g. of Container Runtimes:  1) CRI-O: lightweight Docker alternative, providing CRI compatibility for runc and Kata containers  2) containerd: Cloud Native Computing Foundation (CNCF) project |
| K8s Security | | Security philosophy based on 4Cs model of cloud native security: Cloud, Clusters, Containers, Code | |
| Authentication: verifying identity of a K8s API client  1) Client certificates: client provides an X509 client cert, kubectl uses this mtd  2) Bearer tokens: client passes a bearer token in an HTTP request header  3) OpenID connect tokens: OAuth2 integration w external identity providers. Uses a JSON Web Token (JWT) signed by the provider. JWTs include signed identity data about the user  4) Authenticating Proxy: trusted proxy validates authentication data provided via HTTP headers  5) Anonymous: no authentication | |
| Authorization: whether or not a user has permission to perform a specific operation  1) Node: special use case, grants permissions for kubelet to work w resources on a specific Node  2) Attribute-Based Access Control (ABAC): define permissions through use of policies that use attributes to refer to users, resources, …  3) Role-Based Access Control (RBAC): assign granular permissions to roles. Then assign roles to user  4) Webhook: K8s API server reach out to an HTTP service to determine if permission is allowed. Use this to implement a fully custom authorization scheme | |
| Open Policy Agent (OPA) Gatekeeper: create and enforce policies that control what can and cannot be done in your K8s env  - Validates incoming requests according to the policies you created  - E.g. cannot create a Pod w/o attaching a descriptive label  - If user attempts to do something that violates the policy, API will deny the request  - OPA Gatekeeper is a generalized tool that exists outside of K8s, K8s simply use it to provide a framework for policy enforcement | |
| K8s Networking | | | Cluster Network: K8s uses a virtual network to allow Pods to communicate within the cluster  - Each Pod has its own IP address unique to the cluster  - Pods can communicate transparently, even if they are on diff Nodes |
| Cluster includes a Domain Name Server (DNS) - Allows containers to discover Services within the cluster using hostnames  - K8s containers auto configured to use the cluster DNS server |
| Network Policy: k8s object that allows you to control what network traffic is allowed within the cluster network  - Each Network Policy selects which Pods it applies to  - Policy provides rules that define what incoming and/or outgoing traffic is allowed  - Pods are non-isolated by default, i.e. all network traffic is allowed  - If a Network Policy selects a Pod, the Pod becomes isolated. Only traffic allowed by the Network Policy is allowed. |
| Services | | | Services: expose an app running on a set of replica Pods as a network service  - Select a set of Pods and expose them as a single network entity  - Different Service types for diff use cases  - Clients communicate w the Service. Traffic dynamically directed to one of the underlying Pod endpoints |
| Service Types: 1) ClusterIP: expose Service on an IP addr internally within the cluster network  2) NodePort: expose Service externally using a node port listening on each Node  3) LoadBalancer: expose Service externally using a cloud provider load balancer  4) ExternalName: creates a DNS record pointing to an external location outside the cluster network. Makes it easy for Pods to access an external URL or IP addr |
| Headless Services: services w no cluster IP addr  - Can be used to interface w service discovery mechanisms without actually proxying traffic |
| Services w/o selectors: can create Service w no selector to select backend Pods  - These don’t automatically create endpoints. You need to create any endpoints manually |
| Service Discovery: auto detect or locate app services on a network  1) Cluster DNS: use cluster DNS to reference a Service by hostname. Traffic auto routed to a backend Pod  2) Env variables: k8s auto adds env variables to each container that provide info about Services |
| Ingress: k8s obj that manages external access to apps within the cluster  - Offer additional functionality like load balancing, SSL termination (compared to NodePort, LoadBalancer)  - Ingress routes traffic to a Service on the backend |
| Service Meshes | Service Mesh: manages communication btw app components, often adding additional functionality like logging, tracing, or encryption  - Service Mesh is diff from the k8s Service resource | | |
| Service Mesh Architecture  - Apps components communicate w each other through a service mesh of proxies (sidecars) deployed alongside each component  - Sidecar proxies add the additional functionality provided by the service mesh  2 main components: 1) Service Proxy/Data place: proxy sidecar containers in each Pod. Network traffic flows through them  2) Control plane: controls, configures, and coordinates the data plane proxies | | |
| E.g. of service mesh  1) Linkerd: Open source Cloud Native Computing Foundation (CNCF) project. Metrices, mTLS, more  2) Consul Connect: Build on HashiCorp Consul. Connection authorization and mTLS encryption  3) Traefik Mesh: Open source, built for ease of use. Traffic control and observability  4) Istio: Open-source, platform independent. Policies, traffic management, metrics  5) Kuma: Open source, works w K8s and VMs. Routing, observability, security  6) F5 NGINX Service Mesh: data plane using NGINX Plus. Traffic handling, blue/green deployments, zero-trust mTLS | | |
| Service Mesh Interface (SMI): standard interface for service meshes in k8s  - Configure any SMI-supporting service mesh using custom k8s resources via the k8s API | | |
| K8s storage | Volumes: provide external storage to your k8s containers to store app data | | |
| Persistent Volumes: treat storage resources as dynamically consumable, similar to how k8s treats resources like memory and CPU, i.e. actual storage  Persistent Volume Claim (PVC): binds to a PersistentVolume and allows you to mount the storage resources inside a Pod i.e. user request for storage  So Pod -> Volume -> PVC -> PersistentVolume | | |
| Reclaim Policies: what happens to storage resource when PVCs are deleted  1) Retain: reclaim manually. 2) Recycle: auto reclamation via a simple data scrub  3) Delete: underlying storage resource is deleted (only for cloud storage) | | |
| Rook: storage orchestrator tool that integrates w k8s  - Automate storage management w self-managing, self-scaling, self-healing storage services | | |
| ConfigMaps: store config data like config values, config files, secure credentials & pass it to containers  Secrets: use for sensitive data like passwords or API keys  - By default, Secret data is NOT encrypted, just base64-encoded  - Make config/secrets data unchangeable with *immutable: true* | | |
|  | | |  |

|  |  |
| --- | --- |
| Cloud Native Architecture | |
| Intro | E.g. of Cloud Native Architecture: (helps remove roadblocks to innovate)  Loosely coupled microservices, DevOps, Operational automation, Serverless, Orchestration |
| Autoscaling | Autoscaling: auto assign more or fewer compute resources to an app/system in response to real-time needs |
| Cost advantages of autoscaling:  - scale too low: affect reliability and performance. – too high: affect cost |
| Vertical: add more compute power  - Add additional PCU or memory to cluster Nodes  Horizontal: add more instances of an app (more replicas Pods)  - Add new Nodes to cluster |
| K8s Autoscaling tools:  - Horizontal Pod Autoscaler: monitor resource usage of existing replicas, and creates/destroys replicas when needed  - Cluster Autoscaler: adds and remove Nodes from cluster based on real-time usage |
| Serverless | Serverless: devs build and run apps w/o worrying about servers and server-related concerns like OS, scaling, …  Servers managed completely by cloud provider |
| Cloud Native Community & Governance | Cloud Native Communities: groups of individuals who work tgt to build and promote vendor-neutral cloud native tech, standards, techniques  E.g. Cloud Native Computing Foundation (CNCF) |
| CNCF Governance:  - Governing Board: guides CNCF as a whole  - Technical Oversight Committee (TOC): admits new open-source projects to the CNCF  - Decisions made through public discussion and voting |
| CNCF Community:  - End User Community: provides feedback from end users  - Special Interest Groups (SIGs): oversee and coordinate needs for specific domains or tech  - Work Groups: specialized groups to accomplish a task or define a standard |
| Organizational Personas | Organizational Personas: generalized roles that interact w cloud native tech in diff ways  - Not necessarily individuals or positions, but roles fulfilled w.r.t cloud landscape |
| 1) Developer: writes app code for end users  2) Ops: builds and maintains infra that runs the code. Deploys new code  3) Site Reliability Engineer (SRE): maintains app reliability and performance. Create and maintain service-level objectives (SLOs), service-level agreements (SLAs), service-level indicators (SLIs)  4) Security and Compliance Engineer: develops and maintains security standards. Ensures apps and infra comply w tech and governmental standards |
| Open Standard | Open Standards: technology specification open to public adoption  - Technologies that support the same open standard can work tgt more easily |
| Open Container Initiative (OCI): organization that creates open standards for container formats and runtimes  - Image-spec: OCI open standard for container image format  - Runtime-spec: OCI open standard for container runtimes  Containerd and CRI-O as their runtime use OCI runtime-spec  - Reference implementation for the OCI runtime-spec is *runc* |
| E.g. of Open Standards: HTML, XML, OCI runtime-spec and image-spec, k8s SMI |
|  |  |

|  |  |  |
| --- | --- | --- |
| Cloud Native Observability | | |
| Telemetry and Observability | | Telemetry: collecting data, such as metrics, log data about a system  Observability: ability to understand and measure the state of a system based upon data generated by that system |
| Container Logs in k8s: standard output and error streams go into the container log  *kubectl logs <pod\_name> -c <container\_name>*  *kubectl top <resource\_type>: top system resource usage* |
| Distributed System Tracing: track requests across a complex app consisting of multiple components and services  - Each request or interaction is tagged w a unique identifier  - Helps to understand what is going on as requests make their way through the entire system  - Trace: data about a request as it moves through the system; a set of related events across multiple components  - Span: a part of a trace, representing the request moving through one segment of the system |
| Prometheus Monitoring | | Prometheus: open-source tool for monitoring and alerting. Primary goal is gathering metric data |
| Prometheus metric type: 1) Counter: single num that can only incr or reset to 0  2) Gauge: single num that can go up or down  3) Histogram: counts observations that fit into configurable buckets (e.g. response times)  4) Summary: similar to histogram, but uses dynamic quantiles over a sliding time window |
| Grafana: build useful visualizations of Prometheus data |
| Cost Management | | Cost Management: taking steps to use cloud more efficiently and limit unnecessary costs  FinOps: practice of using observability to support automation and data-driven decisions to limit cloud cost |
| E.g. – collect data in Prometheus to show there is more than enough capacity to handle current load and scale down  - Gather metrics about compute resource usage in order to choose more efficient cloud services for apps that are not used as much  - Use cluster autoscaler to temporarily scale the cluster up in order to run a large batch processing job |
|  |  | |

|  |  |
| --- | --- |
| Cloud Native Application Delivery | |
| App Delivery Intro | Application Delivery: processes and techniques used to ship new code to customers (aka deployment) |
| Challenges: change brings instability  - Deployment Woes: problems caused by process of deploying code. - Bugs: problems in code itself.  – Configuration: complexity of managing config across envs |
| CI/CD (Continuous Integration/Continuous Delivery)  Automation: consistent, automated processes. Flexible, fluid infra |
| GitOps | GitOps: use Git to manage app infra. - Git is the source of truth for declarative infra/apps  - Automation implements what is in Git |
| Git repo contains files describing the cluster’s desired state  Tool watches the Git repo. Tool auto make changes to implement what is in Git |
| GitOps Tools: 1) Flux: syncs manifests stored in a Git repo w your cluster  - Built on top of GitOps Toolkit. – Written in Go  2) Argo CD: uses GitOps to do Continuous Delivery in k8s. - Includes a GUI. – Written in Go |
| CI/CD | CI: integrating code from multiple developers frequently. – Utilizes automation  - Compiling, integration, tests occur automatically when a dev pushes code changes to a repo like Git |
| CD: deploying code frequently w/o interrupting stability/availability to users. – Uses automation  Sometimes called continuous delivery |
| Frequent deployments, Incremental changes, Automated processes, Reliable rollbacks |
|  |  |