|  |  |
| --- | --- |
| **Principles and roadmap** | |
| **DevOps Cycle**: plan->code->build->test->release->deploy->operate->monitor | |
| **Culture –** Collaborative and customer-centred  **Automation** – Integration testing, deployments, IaC  **Lean** – Agile, scrappy, lean teams to minimize WIP (2-pizza rule)  **Measurement** – Track and measure data to celebrate wins & pre-empt faults  **Sharing** – Teach & learn from each other | |
| **Gene Kim’s 3 ways**:   * Flow - Convert a business hypothesis into a technology-enabled service that delivers value to the customer * Learning – Ensure fast/constant feedback in all stages of the value stream * Feedback – Prioritize organizational learning and safety culture | |
| **Flow**:   * Make work visible * Limit WIP * Reduce batch size and handoffs * Identify and elevate constraints * Eliminate waste in the value stream | |
| **Feedback**   * Monitoring to see problems as they occur * Swarming problems to build new knowledge * Quality at source * Optimize for downstream work centres | |
| **Continual learning and experimentation**   * Improve daily work * Local discoveries to global improvements * Resilience patterns that introduce chaos and tension * Leadership – creating conditions for success | |
| **Microservice**: Anything that provides functionality over a network – logically represents a business capability, otherwise treated as a black box that can be independently developed/deployed | |
| **Microservice architecture**: An assembly of fine-grained services that allow independent, continuous deployment | |
| **Docker** | |
| **Choreography**: Services communicate with each other asynchronously | |
| **Orchestration**: Composite services invoke atomic services to fulfil business functions | |
| **Atomic service**: Provides functionality related to 1 capability. Self-contained and do not depend on any other services | |
| **Drawbacks of microservices**:   * Microservices make things worse, not better, if developers have a poor testing culture * Decomposing a service incorrectly gives a distributed monolith * Distributed systems are inherently more complex | |
| **Docker networking**: All traffic is routed through docker engine. Containers can communicate with each other within the network, but outside applications must use forwarded ports as specified by Dockerfile/docker-compose.yml | |
| **Continuous Integration** | |
| **Test pyramid** (descending order of difficulty, brittleness and cost): -UI testing  -End-to-end testing  -Component testing  -Integration testing  -Unit testing  -Code analysis | |
| **Continuous Deployment** | |
| **Continuous delivery** – releases are automated, deployments are manual **Continuous deployment** – deployments are fully automated | |
| **Benefits**: Lower failure rates, faster feedback, faster flow, reliable releases  **Tradeoffs**: Building binaries for diff platforms, customer perceptions of CD, only works with a testing culture | |
| **Release versioning**: Traditionally semantic versioning <major>.<minor>.<patch>   * Major: breaking, incompatible API changes * Minor: backwards-compatible functionalities * Patch: backwards compatible bug fixes | |
| **Serverless deployment**: Abstracts away infra & resources, invoking microservices implemented as lambda functions.   * Pros: Very lightweight and scalable * Cons: Cold starts, large functions, not suitable for long-running jobs | |
| **Service as a container**: Package as a Docker image and deploy each service as a container. Each service has its own IP and file system   * Pros: Portable, isolated, constrained resources, encapsulation * Cons: Responsibility for administering images and specifying/administering container infrastructure | |
| **AWC ECS concepts**:   * Task definition: Describes containers that form your app – images and resource constraints * Service: Runs and maintains a specified number of tasks (can set auto-scaling as well) * Cluster: Logical grouping of tasks and services. Able to orchestrate containers across multiple EC2 instances | |
| **Deployment patterns**   * Rolling: Gradually replace container instances, specifying a minimum healthy % * Blue-green: Transfer all traffic to new container instances; old instances on ‘standby’ for rollback * Canary: Release to a small subset of users first | |
| **Feature flags**: Allow new features to be enabled/disabled on toggle, promoting easy rollback, graceful degrades, resilience in deploys | |
| **Microservices communication** | |
| **Styles**: 1-1 sync(HTTP, gRPC), 1-1 async(fire & forget, async request/reply), 1-many async (pub/sub) | |
| **Message-oriented middleware**: Acts as a broker for any communication style | |
| **Communication patterns**  1. Orchestration – Stateless composite service that manages atomic services  2. Choreography – Atomic services communicate with each other asynchronously  3. Choreo with process engine – Combines async behaviour & process visibility | |
| **AMQP**: A public message queueing protocol that runs on top of TCP  Publisher -> Exchange -> Queue -> Subscriber. Publishers publish message to an exchange, exchanges are bound to queues using a routing key (direct/fanout/topic), subscribers consume messages from queues. | |
| **Saga pattern**: Create a set of compensating transactions for every local transaction that a microsvc makes. Since microservices are not isolated, concurrency is still an issue. | |
| **Solutions**: RabbitMQ (message broker for AMQP), prog. language implementations (pika, etc.) | |
| **API gateways** | |
| **Strangler fig pattern**: Gradually decouple the system from top-down, starting with edge services and working to core. Transform -> Co-exist -> Eliminate | |
| **API gateway patterns**:  1. Composition – aggregating requests to atomic svcs  2. Protocol translation  3. Edge functions (auth, rate limiting, caching, metric, logging), consider *separation of concerns* | |
| + Encapsulation/façade  + Reduces network overhead  + Simplify client-side interactions  + Loose coupling | - More management  - Must be highly available  - May cause development bottleneck  - No clear ownership |
| **General purpose**: One API gateway for all frontend clients  - General purpose, multiple responsibilities, no clear ownership (against DevOps style) | **Backends for frontends (BFF)**: Each frontend (mobile/web/native) has its own gateway  - Frontend oriented, more separated responsibilities, easier tracking, possible code duplication |
| **Solutions**: AWS API gateway, Kong, DIY (nginx, zuul, etc.) | |
| **Kubernetes** | |
| **Declarative model**: Use manifest files (YAML) to specify desired state, and k8s takes actions to achieve that desired state | |
| **Node**: Contains services required to run pods  **Kubelet**: Agent that runs on a node, ensuring that pods are running and healthy  **kube-proxy**: Allows communication over virtual network  **Pod**: A basic unit of deployment, with at least 1 container sharing an internal IP address and storage volume. Meant to be ephemeral and replaced  **API-server**: Allows dev to interact with k8s deployment through UI, CLI or REST calls  **Controller manager**: Keeps track of cluster state and makes adjustments if necessary  **Scheduler**: Watches for newly created pods and selects nodes for them to run on | |
| **Service**: Abstraction that provides a static network location that exposes some pods. Internal addresses of Pods may change, but service ip/port does not change   * *ClusterIP*: Service reachable on an internal IP * *NodePort*: Service reachable on Node’s IP at a static port, externally reachable at <NodeIP>:<NodePort> * *LoadBalancer*: Service exposed externally using cloud provider’s load balancer (1-to-1) | |
| **Ingress**: Exposes multiple Services through a single cloud load balancer  **Ingress controller**: Implementations include cloud providers, nginx, etc | |
| **Rolling updates**: Default rolling update has a limit of 25% unavailable pods/extra pods  **Auto-scaling rules** can be set based on CPU usage, min or max replicas | |
| **Monitoring** | |
| **Telemetry**: Automated process of collecting/transmitting metrics from remote points to monitoring systems. 1-Instrument systems, 2-Collect metrics from systems, 3-Monitor through alerts and visualizations | |
| **Challenges**: Number of components, data quantity, silos, maintenance burden | |
| **What to monitor**?:  *Work metrics* – Throughput, success, errors, latency  *Resource metrics* – Utilization, saturation, availability  *Events* - Code changes, alerts, scaling events  *CI/CD* – Time taken for pipeline stages, pass/fail events | |
| **Log levels**: DEBUG -> INFO -> WARN -> ERROR -> FATAL, severity level dictates incident response | |
| **Self-service**: Metrics should be easily accessible, “info radiators” w/o privileged access | |
| **Analysis**: Through statistical operations or anomaly detection (independent of probability distribution) | |
| **Prometheus**: Centralized telemetry software, stores metrics in time series database, taken from service endpoints in a PULL model. Query language PromQL allows selection and aggregation of data. Data can then be visualized through software like *Grafana* | |
| **Infrastructure as Code** | |
| **IAC tools**:   * Shell scripts * Configuration management: Install/manage software on servers * Provisioning: Create ephemeral resources (servers, brokers, security groups) | |
| **Terraform**   * init: Initialize current working directory and prepare files for use w Terraform * fmt: Rewrite .tf files to a canonical style * plan: Reads remote state, updates local state, compares local state with .tf files and proposes changes to make remote object match desired configuration * show: Displays local state snapshot * destroy: Destroys remote infra managed by Terraform | |
| **Importing infrastructure**   * Declare a blank resource type and resource name * terraform import resource\_type.resource\_name <remote.id> * Copy local state into configuration file, making sure to delete any ephemeral resource fields (e.g. ARN, ID, IP address) | |