CS3219
AY23/24 Sem 1
github.com/SeekSaveServe

Lectures

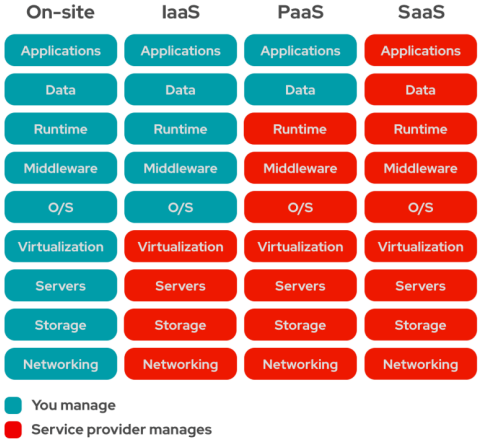
L0 and L1

Software engineering is systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software

Software Crisis 1.0 and 2.0
- (1.0) development of more powerful machinary
- (2.0) growing demand for more complex software due to hardware advances, cost reduction, data reduction, data availability, device proliferation and the rise in consumer technology

Software at the Edge
- Balance between the demands of centralized computing and localised decision making
- Cloud-based: latency that makes it unsuitable for real-time applications
- Edge: Limited computation power and power source

Cloud Computing
- Software infrastructure hosted on an external data centre
- Cloud-enabled: Legacy enterprise applications designed for local datacentres but modified to run on the cloud
- E.g IAAS, PAAS, SAAS



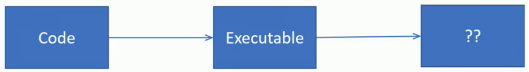
https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas

Cloud-native applications
- The approach to build, deploy and manage modern applications in cloud computing environments
- Characteristic features:
  1. Immutable infrastructure
  2. Microservices-based applications
  3. API driven
  4. Service mesh
  5. Containers
  6. Dynamically managed
- Monolith: There will be API end points to gain certain functionalities
- Microservice model (cloud-native) is similar to monolith except that functionalities are provided for you

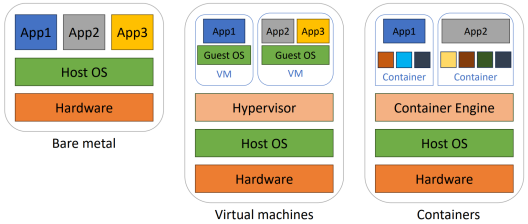
Deployment Considerations

Quality Attributes
- Availability, performance, security, usability, interoperateability, scalability, maintainability, portability, reusability

Issues
- Exploit hardware advances
- Cater to devices
- Network devices, users, applications
- Manage Data
- Large-scale content delivery, quality of service, number of end points
- Shift towards could-nativity
- Heterogeneity platforms, interoperatbility of different OS/Browser/Platform combinations



Code to exectuable - Bare metal
- catering to specific platforms
- customized build and linking
- Factors: availability of libraries and dependencies
- Cons: Potentially wasted hardware resources, cost, developer productivity, scalability (hardware, software)
- Pros: Complete control, physical isolation



Virtual Machines
- Improve resource utilization and cost
- Flexible (not limited to hardware like baremetal) and scalable, runs on different hardware
- Still runs a full OS
- Side-channel attacks: VMs share the same physical hardware, so they can be attacked by exploiting the shared resources
- Noisy neighbor: VMs compete for resources, so one VM can hog the resources and affect the performance of other VMs

Containers
- Lighter than VM (only has OS processes and libraries)
- Better utilization of hardware resources
- Allows for rapid deployment, runs everywhere
- Granular and controllable
- Abstracts away the hardware (managed by container engine) so it can run on any hardware – improves performance
- Reproducible, isolation, security
- Cons: not suitable for performance critical software since containers goes through more layers than VMs or bare metal

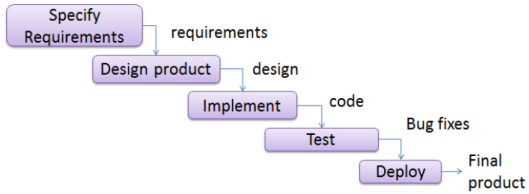
Containerised Deployment
- Easy integration ofthe internet and related advances (build cloud-native apps)
- Include runtime with code (caters to heterogeneous platforms and achieves interoperability and portability)
- Supports dependenct and change management (improves maintainability and portability)
- Environment management (dockerfile provides the environment description)
- Reproducible (guaranteed to be identical on container-capable systems)
- Isolation and security (avoid conflicting dependencies and provides sand-box for execution)
- Quick to launch
- Support DevOps best practices
- Can be used with orchestrators (e.g Kubernetes)

Container Vs orchestrator
- Containers: Provide platform for building and distributing services. Not good for runing complex applications, often requiring multiple containers that each do specific tasks
- Orchestrators: Integrate and coordinate containers, providing scaling of deploymend based on demand, fault tolerance, and communication among containers

Serverless
- Cloud-native deployment model, servers and underlying infrastructure still exist but are abstracted away
- Developers package containers, apps respond to demand and there are no cost when idle
- Serverless allocate resource dynamically for the developer
- Good for stateless applications (the state does not matter outside of the current execution)

L2: Software Development

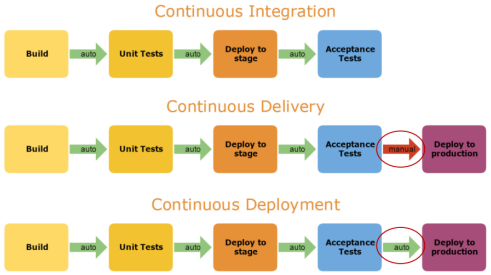
Waterfall



- Good for stable well-understood specifications, familiar domain and solution
- Variation: have feedback loops

Scrum

CI CD Delivery



- CI is a dev practice that requries devs to integrate code into a shared repository several times a day
- Each check in is autmatically verified, allowing for early error detection
- Continuous Delivery Vs Deployment: Software can be Vs automatically released into production at any time
- **benefits**
  1. low-risk releases
  2. faster time to market and early feedback
  3. higher quality and lower cost
- **Blue-green deployment**: Maintain two copies of production environment, one active and one inactive. When a new version is released, it is deployed to the inactive environment and tested. Once it is verified, the inactive environment is switched to active

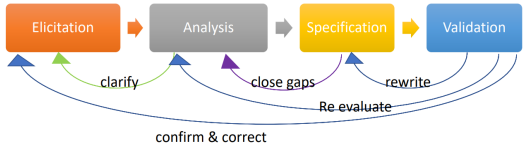
DevOps
- Communication and collaboration
- Infrastructure as code
- monitoring and logging and integration
- Software Reliability Engineer: Create software that improve the reliability of sys in prod, fix issues, respond to bugs, and on-call

Recent trends
- AI to automate CICD
- Pattern detection in code base to detect vulnerabilities
- Collaborating with AI to do code sequence prediction and generation, brainstorm solutions, rapid prototyping
- **Risks**
  1. Garbage in garbage out
  2. Traceability of the source
  3. Data safety
  4. Susceptible to vulnerabilities
  5. Increasing technical debt

L2: Software Requirements

Definition
- Capabilities needed by a user to solve a problem or achieve an objective
- Capabilities must be met or possessed by a system component
- documented representation of the above
- **Usage centric:** As a user, I can ....
- **Product centric:** [Refinement] the system will support the following ...

Four phases of requirements dev



- Elicitation: discover requirements (docs, interviews, surveys, event-response tables, prorotyping, observations)
- Analysis: negotiate requiements, identify gaps
- Specifications: written and illustrated requirements for comprehenon
- Validation: Confirm correct set of requirements that will enable developers to build a soln

SRS

SRS Vs Product backlog
- Both informs whats to be done
- Product backlog faciliates with planning while STS is an in-depth

Qualities of strong SRS

FR Vs NFR

- FR speciefies what the system should do
- E.g. The passenger shall be able to print boarding passes
- NFR describes something not directly related to the functionality of the system
- E.g. Mean time between failure $\geq$ 900ms
- NFR affects system choice and design

Software Quality Attributes