Introduction to Jenkins

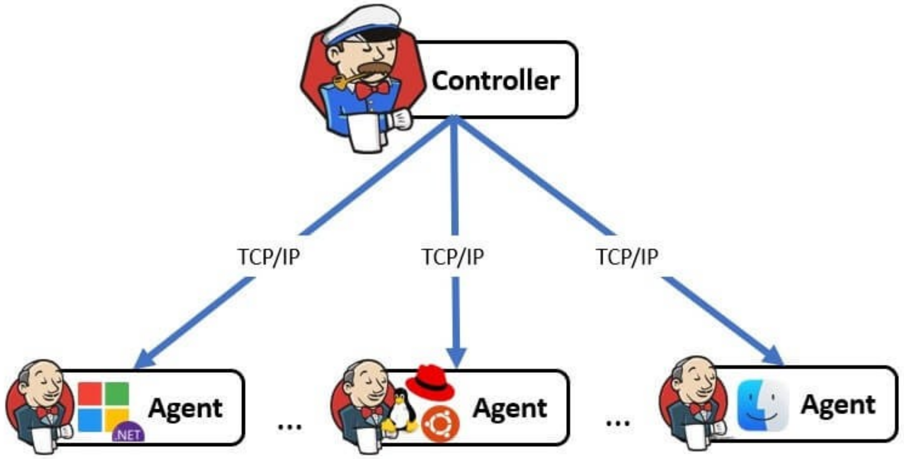
Module 9: Controller/Agent Architecture

Topics

- Static vs. ephemeral agents
- SSH, inbound, and cloud agents
- Docker agents and container-based builds
- Kubernetes agents with Jenkins Kubernetes Plugin
- Workload distribution and performance

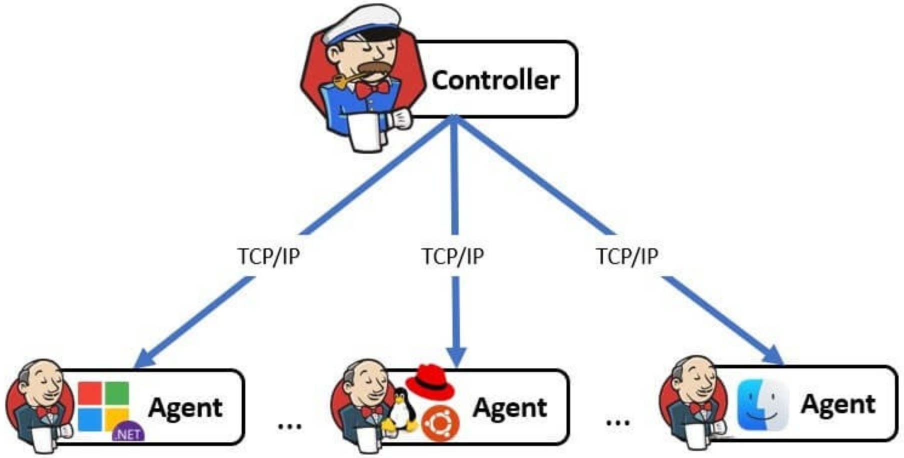
Jenkins Agents Recap

- The Jenkins controller is the brain of Jenkins
  - It is responsible for
    - *Scheduling jobs*
    - *Managing configuration*
    - *Storing build metadata*
    - *Hosting the UI and REST API*
    - *Orchestrating agents*
  - The controller should not be subject to workloads that might degrade the performance of these responsibilities



Jenkins Agents Recap

- Agents are worker nodes that
  - Execute pipeline steps
  - Run build tools
  - Compile code
  - Run tests
  - Perform deployments
- Agents
  - Connect to the controller
  - Execute work
  - Report results back



Static and Ephemeral Agents

- Static agents are:
  - Long-running machines
  - Permanently attached to Jenkins
  - Often manually provisioned
  - Examples:
    - *A fixed Linux VM*
    - *A Windows build server*
- Pros
  - Simple to understand
  - Predictable environment
- Cons
  - Tool drift over time
  - Manual maintenance
  - Limited scalability

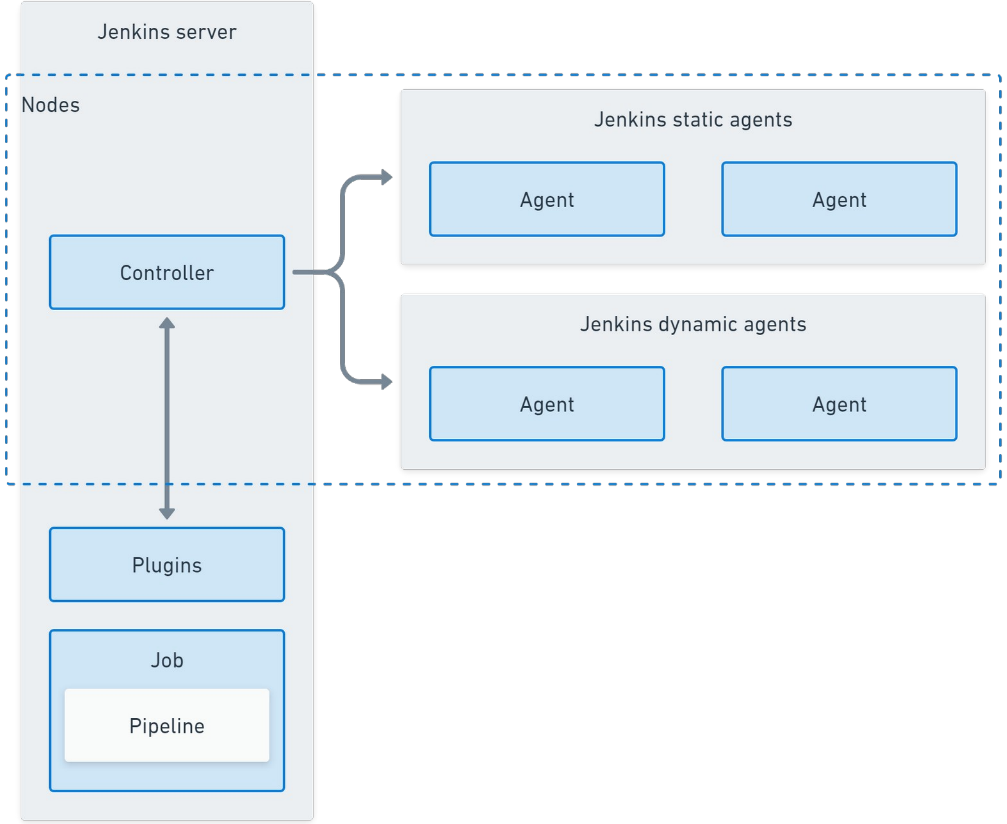


Image Credit: https://octopus.com/devops/jenkins/

Static and Ephemeral Agents

- Ephemeral agents are
  - Created on demand
  - Used for one build
  - Destroyed afterward
  - Examples:
    - *Docker containers*
    - *Kubernetes pods*
    - *Cloud-provisioned VMs*
- Pros
  - Clean environments
  - Consistent builds
  - Easy scaling
- Cons
  - Slight startup overhead
  - Requires infrastructure integration

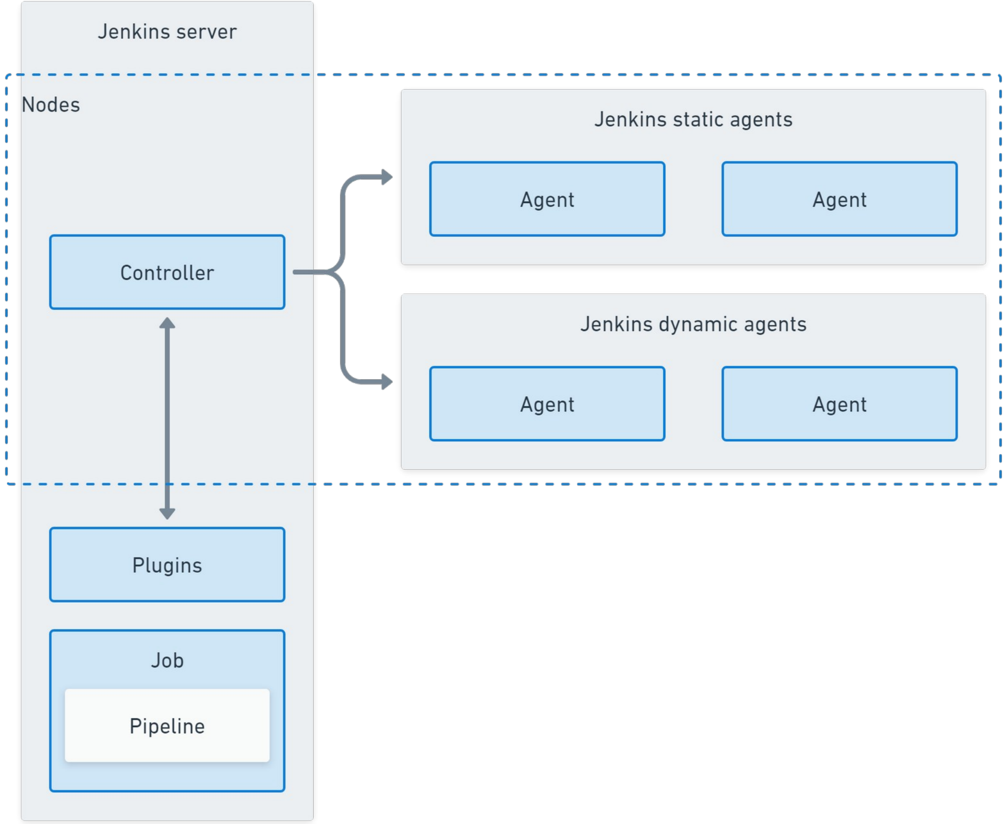


Image Credit: https://octopus.com/devops/jenkins/

Static and Ephemeral Agents

- Static agents use cases
  - Small teams
  - Low build volume
  - Legacy environments
  - Proof-of-concept Jenkins installs
  - Regulated environments with fixed infrastructure

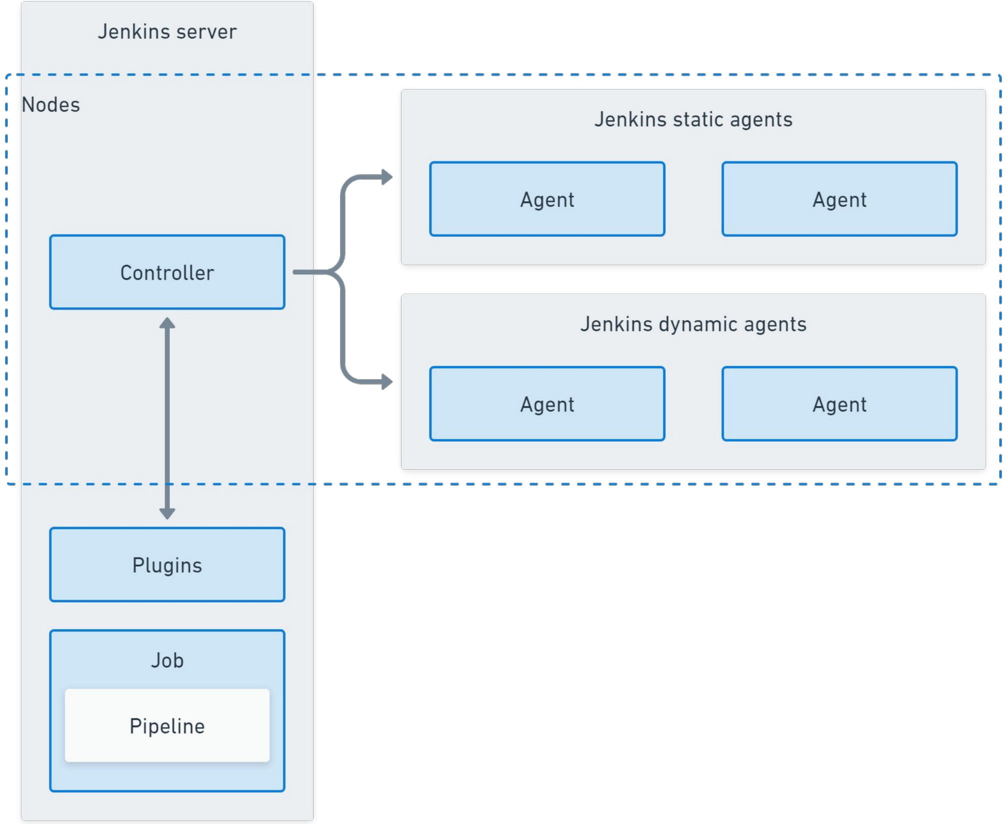


Image Credit: https://octopus.com/devops/jenkins/

Static and Ephemeral Agents

- Ephemeral agents use cases
  - Medium to large organizations
  - Multiple teams using Jenkins
  - High or bursty CI workloads
  - Cloud-native environments
  - Security-sensitive pipelines
  - Long-term Jenkins adoption

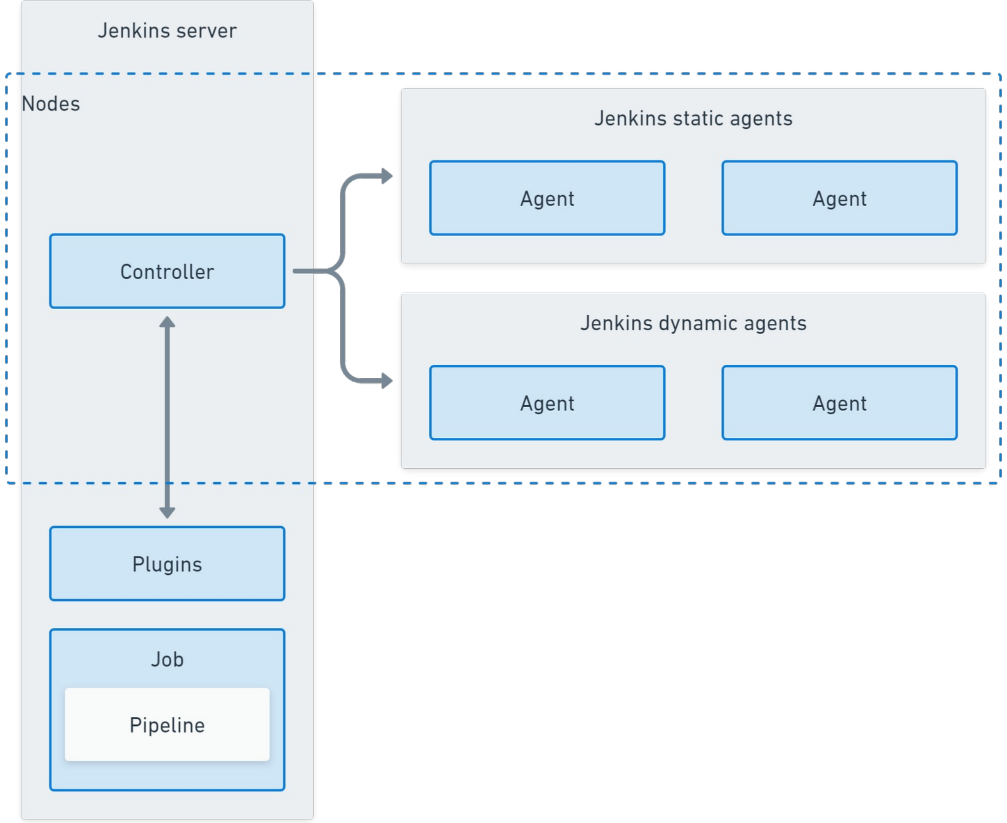
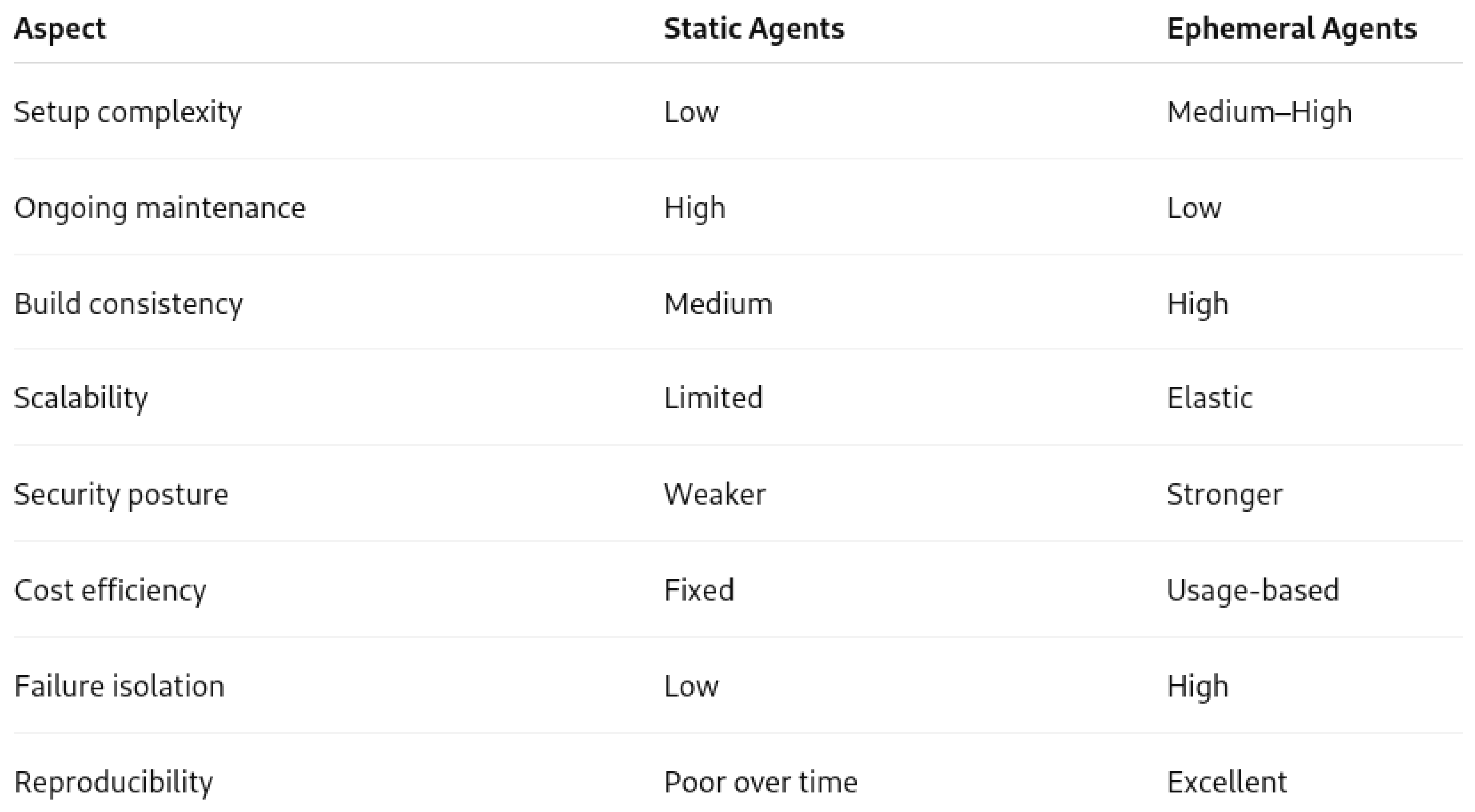


Image Credit: https://octopus.com/devops/jenkins/

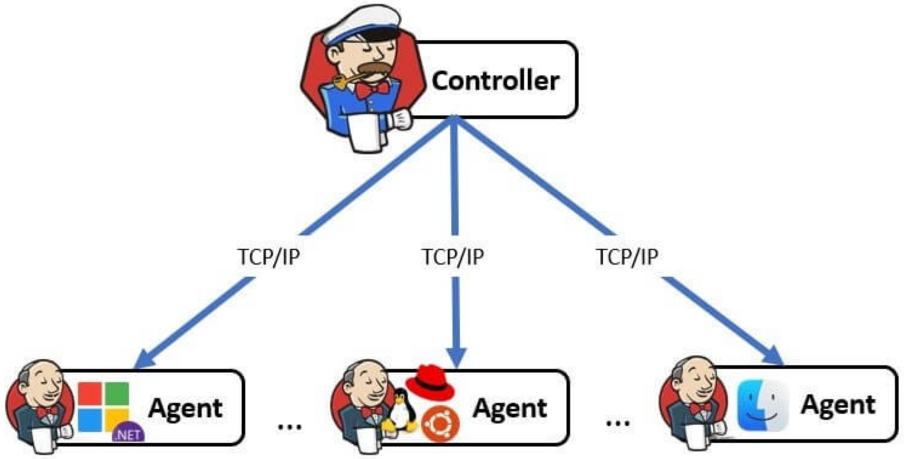
Static and Ephemeral Agents



Aspect	Static Agents	Ephemeral Agents
Setup complexity	Low	Medium–High
Ongoing maintenance	High	Low
Build consistency	Medium	High
Scalability	Limited	Elastic
Security posture	Weaker	Stronger
Cost efficiency	Fixed	Usage-based
Failure isolation	Low	High
Reproducibility	Poor over time	Excellent

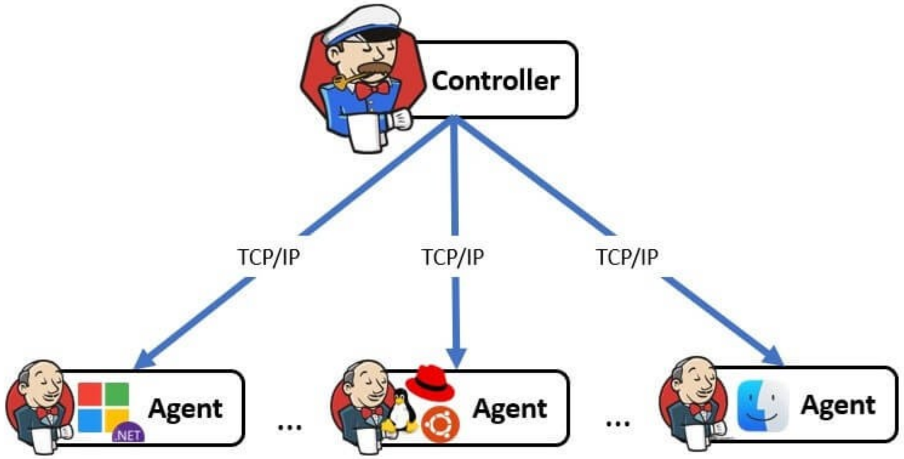
SSH Agents

- An SSH agent
  - Runs on a remote machine
  - Is accessed by the Jenkins controller over SSH
  - Original Jenkins agent model, especially in Unix/Linux environments
- Connection Flow
  - Jenkins controller has SSH credentials
  - Controller initiates an SSH connection to the agent machine
  - Jenkins launches an agent process remotely
  - Agent listens for work from the controller
  - Build steps execute on the remote machine
  - Results are sent back to the controller



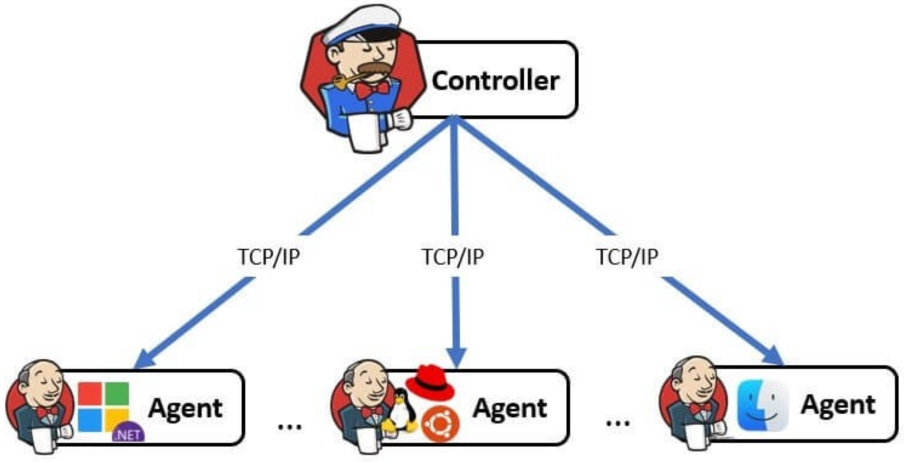
SSH Agents

- The SSH agent is a separate host
  - Usually a
    - *Physical server*
    - *Virtual machine*
    - *Cloud VM*
- Runs its own OS and tool chain
  - Protects the controller from the build environment
  - Allows workload isolation
  - Enables horizontal scaling



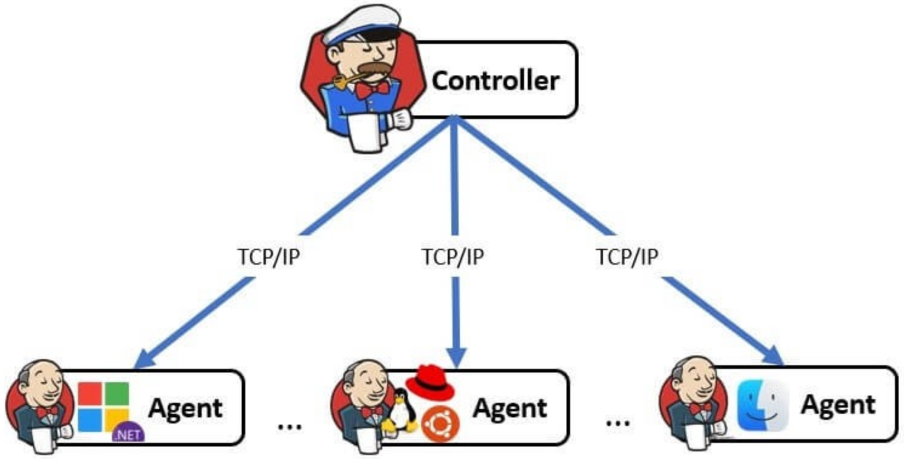
SSH Agents Use Cases

- Existing servers
  - SSH agents are ideal when an organization already has:
    - *Dedicated build servers*
    - *Legacy infrastructure*
    - *Long-running VMs*
  - Instead of replacing these machines Jenkins simply connects to them
  - Reduces
    - *Migration effort*
    - *Initial cost*
    - *Disruption*



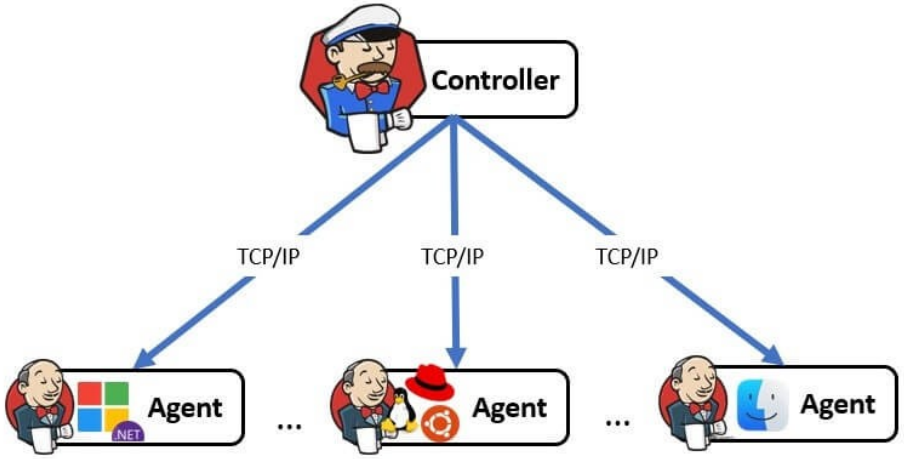
SSH Agents Limitations and Risks

- Configuration drift
  - Because SSH agents are long-lived
    - *Tools get upgraded manually*
    - *Builds change behavior over time in response to environment changes*
    - *Reproducibility suffers*
- Maintenance burden
  - Ops teams must
    - *Patch OS*
    - *Upgrade tools*
    - *Clean workspaces*
    - *Rotate SSH keys*
  - This work grows with
    - *Agent count*
    - *Team size*



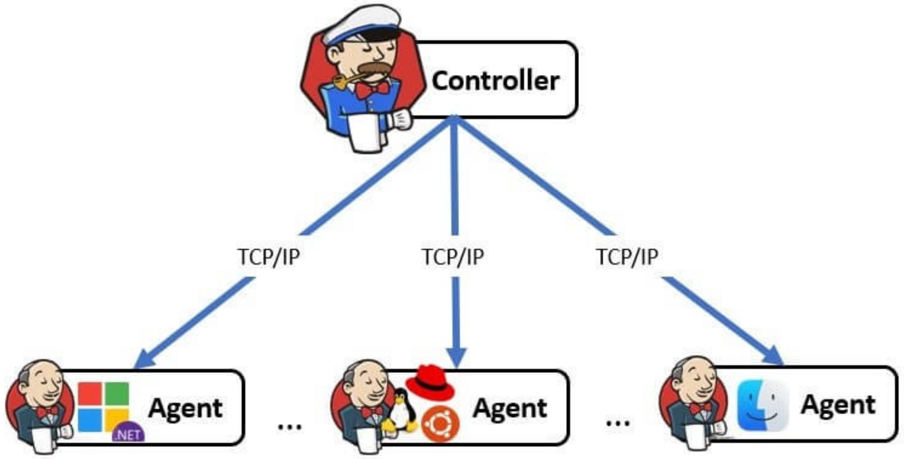
SSH Agents Limitations and Risks

- Scalability constraints
  - Agents are finite
  - Build spikes cause queues
  - Adding capacity is slow
  - SSH agents do not scale automatically
- Security considerations
  - SSH keys must be protected
  - Compromised agents pose risk
  - Credentials may linger on disk
  - This makes SSH agents less attractive for
    - *High-security environments*
    - *Regulated workloads*



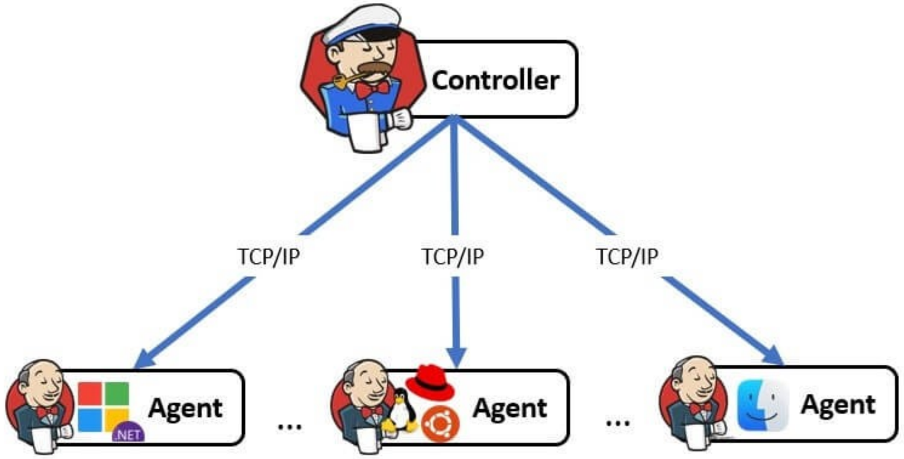
SSH Agents

- SSH agents are appropriate when
  - Infrastructure is static
  - Network is trusted and stable
  - CI workload is predictable
  - Cloud or container adoption is low
- SSH agents are not ideal for
  - Highly elastic CI
  - Short-lived builds
  - Modern cloud-native pipelines



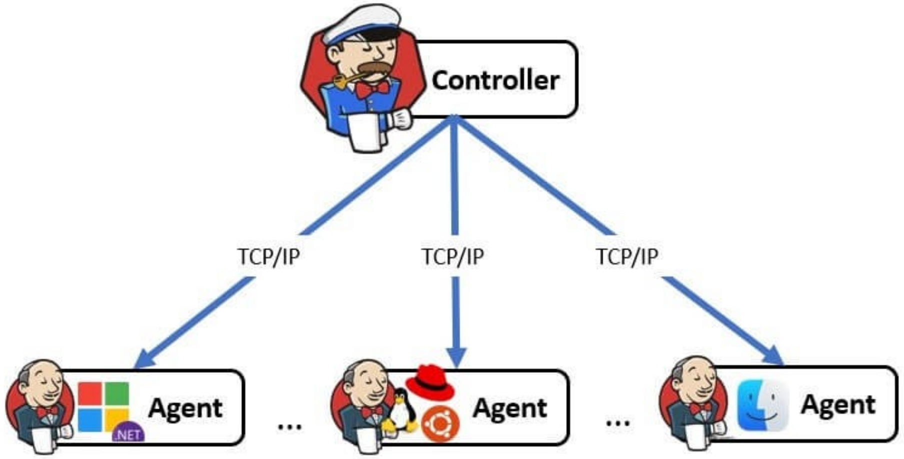
Inbound Agents

- An Inbound agent
  - Initiates the connection to the Jenkins controller
  - Connects using a secure inbound protocol
  - Waits for work from the controller
  - This reverses the SSH model
  - The agent connects to Jenkins, not the other way around
  - Also called JNLP agents
    - *JNLP = Java Network Launch Protocol*
    - *Historically used by Java-based agents to connect to the controller*



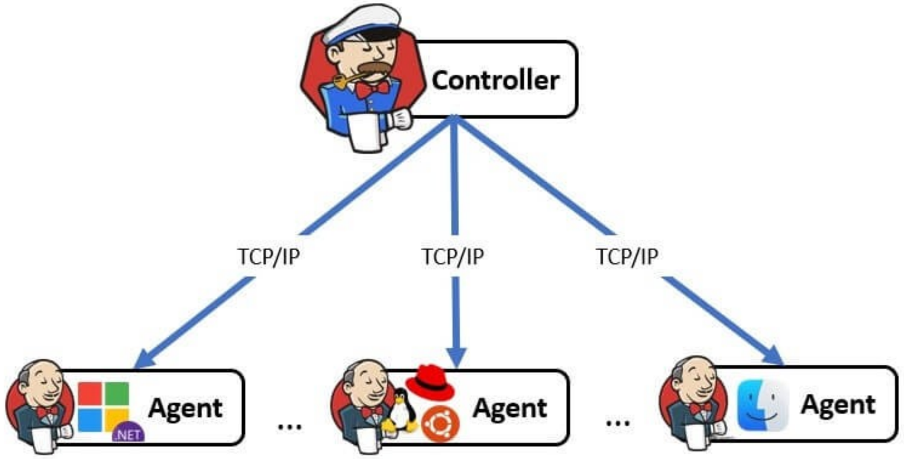
Creating Inbound Agents

- An agent node is created in Jenkins which generates
  - A secret
  - A JNLP URL (connection info)
- The agent machine runs
    - *java -jar agent.jar -jnlpUrl <url> -secret <secret>*
  - The agent initiates the connection to the controller
  - Jenkins accepts it and schedules jobs on the agent



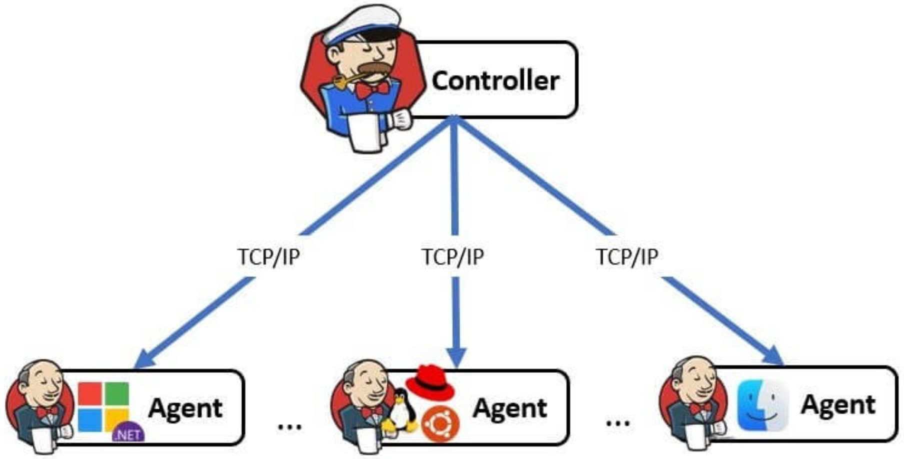
Inbound Agents

- Connection flow
  - Agent process starts on a machine
  - Agent opens an outbound connection to the controller
  - Controller authenticates the agent
  - Agent registers and waits for work
  - Build steps execute on the agent
  - Results are sent back to the controller
  - No inbound firewall rules are required on the agent side



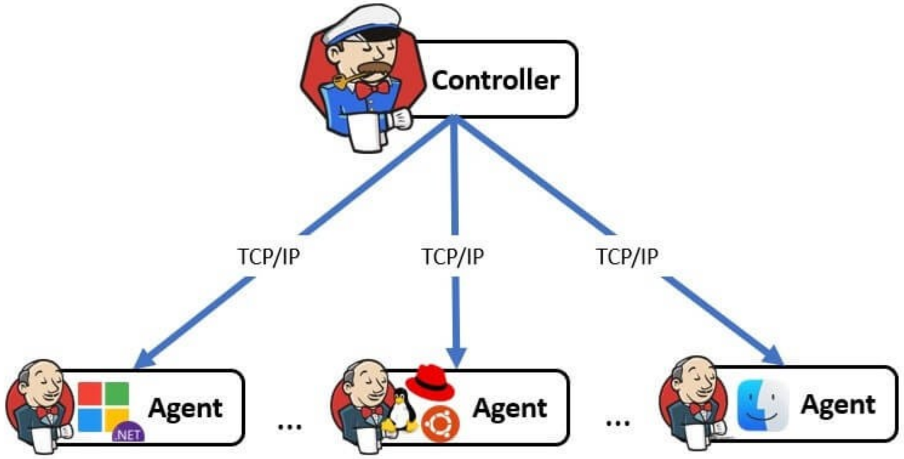
Inbound Agents

- Inbound agents connectivity
  - Use outbound connections only
  - Work through strict firewalls and NAT
  - Avoid opening SSH ports
- Ideal for
  - Locked-down corporate networks
  - Cloud environments with strict security controls
  - Environments where inbound access is prohibited



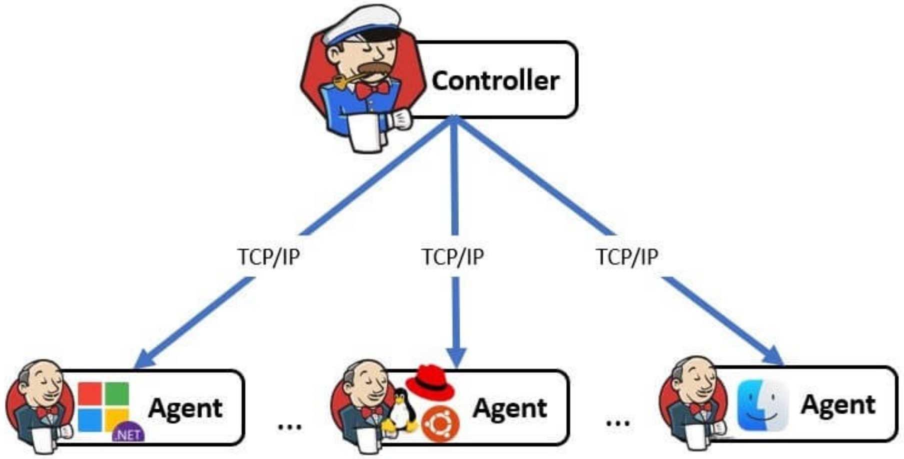
Inbound Agents

- Inbound agents are widely used in
  - Cloud VMs
  - Container-based agents
  - Kubernetes pods
  - Auto-scaled environments
  - Cases where the agent is ephemeral
- Default choice for modern Jenkins deployments



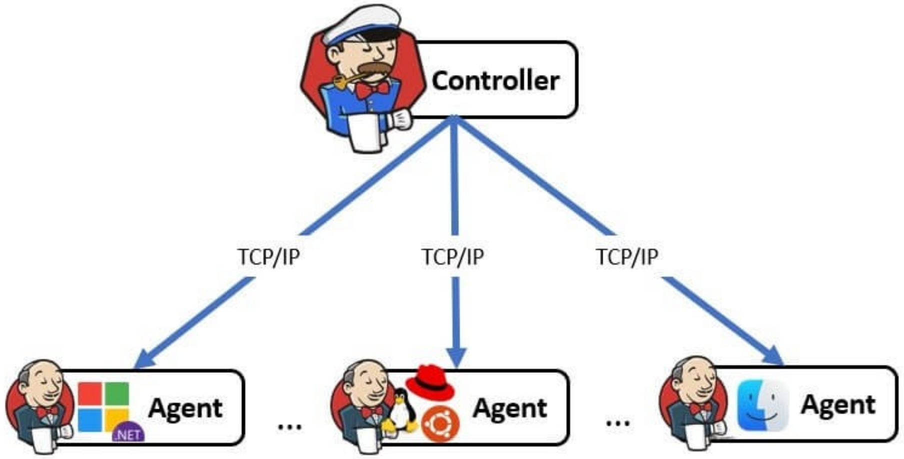
Inbound Agents Use Cases

- Locked-down networks
  - Inbound agents work where
    - *Agents cannot be reached directly*
    - *Security policies forbid inbound access*
    - *Only outbound HTTPS is allowed*
  - Common in
    - *Financial institutions*
    - *Regulated industries*
    - *Zero-trust networks*



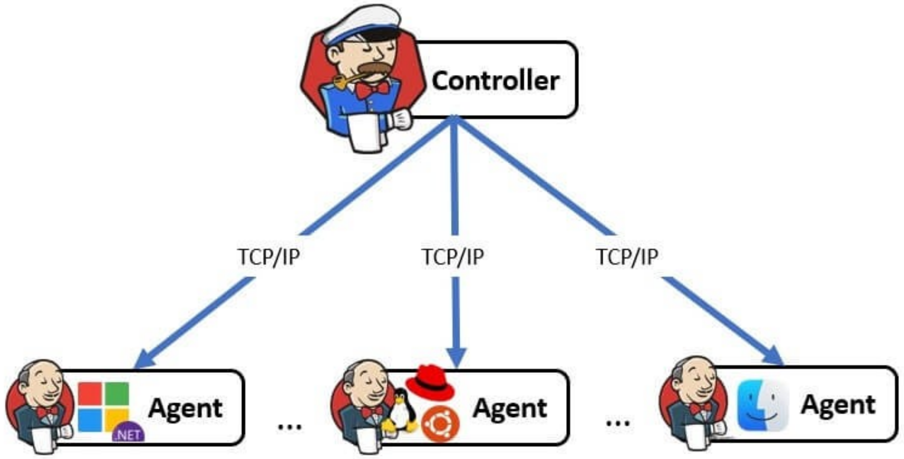
Inbound Agents Use Cases

- Agents started dynamically
  - Inbound agents are ideal when
    - *Agents start and stop automatically*
    - *Infrastructure is short-lived*
    - *Jenkins should not manage host access*
  - Examples
    - *Docker containers*
    - *Kubernetes pods*
    - *Auto-scaled cloud instances*



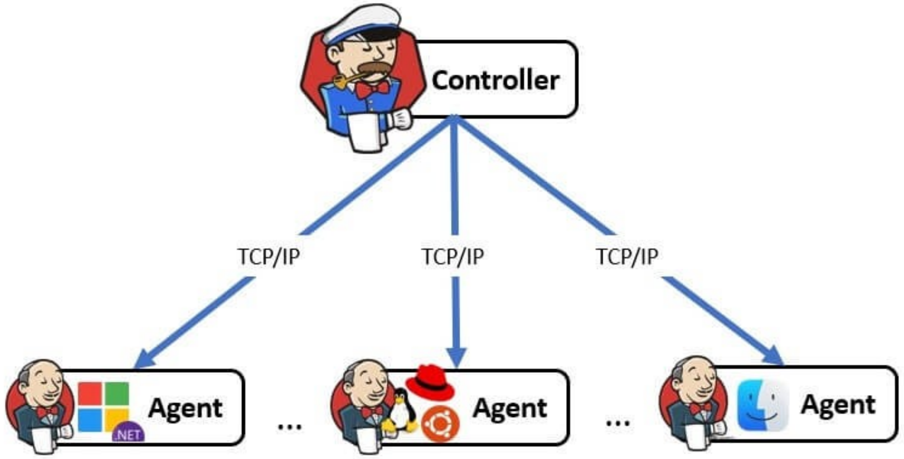
Operational Advantages

- Security
  - No SSH keys on the controller
  - Reduced attack surface
  - Easier firewall configuration
- Scalability
  - Agents come and go
  - No static registration required
- Flexibility
  - Works across networks and clouds



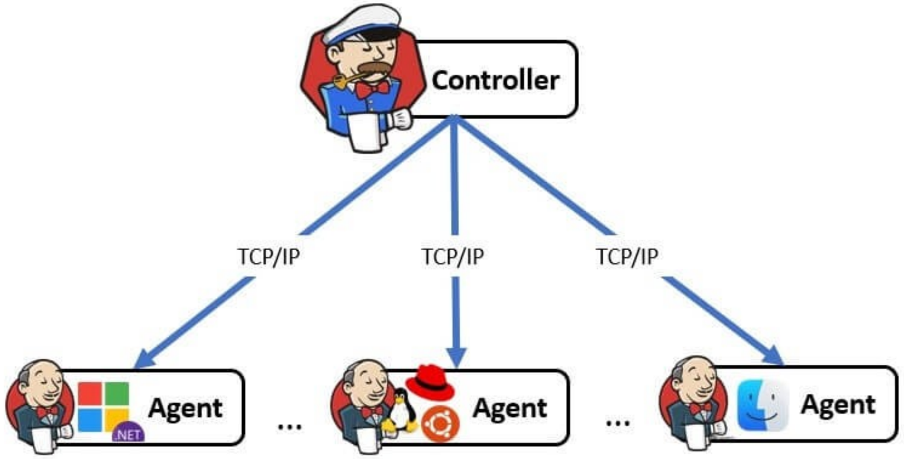
Operational Tradeoffs

- More moving parts
  - Requires agent launch logic
  - Requires correct agent configuration
- Harder to debug
  - Connection issues can be less visible
  - Logs may be distributed



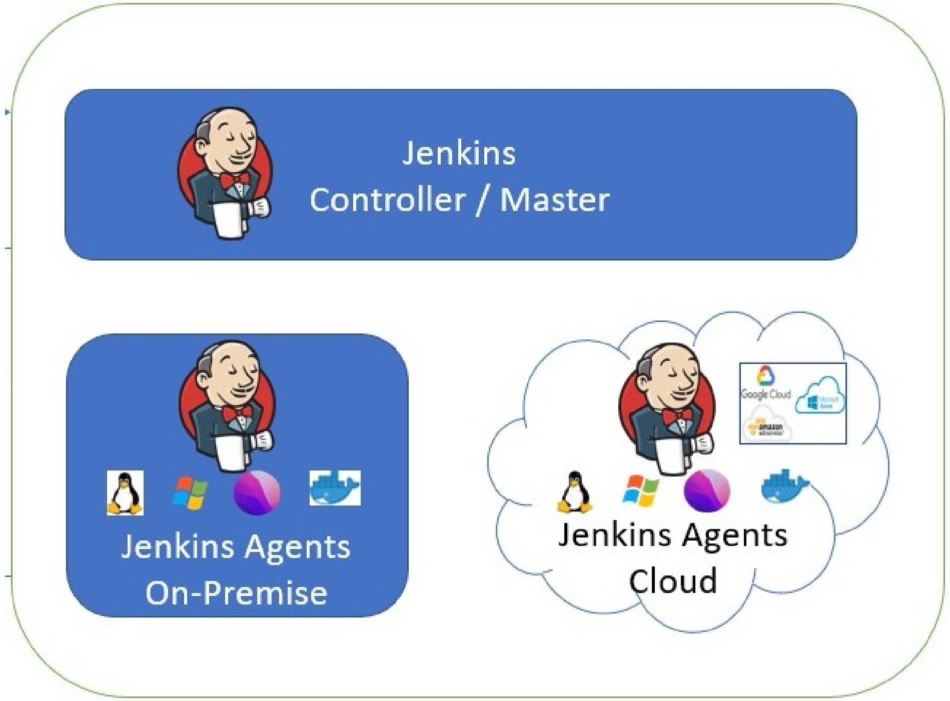
Inbound Agents

- Inbound agents are the right choice when
  - Agents are ephemeral
  - Networks are restrictive
  - Jenkins runs in the cloud
  - Kubernetes or container agents are used



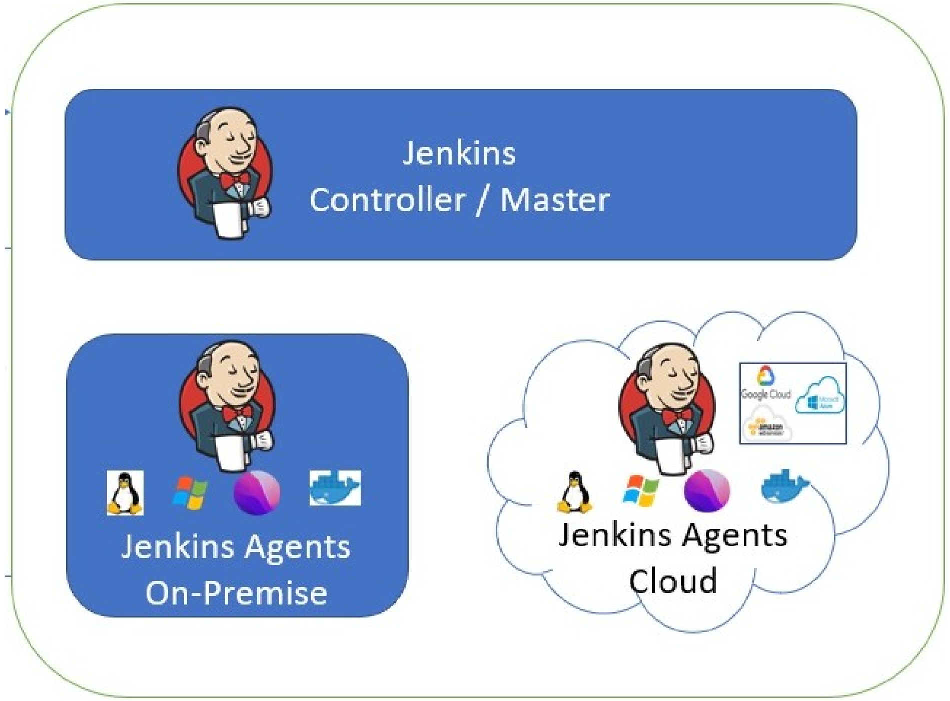
Cloud Agents

- Cloud agents are
  - Provisioned automatically by Jenkins
  - Created only when there is work to do
  - Destroyed when no longer needed
  - Managed through cloud provider APIs
- Allow Jenkins to scale capacity dynamically
  - Rather than relying on permanently running build machines
- Cloud agents turn Jenkins from a fixed capacity system into an elastic service



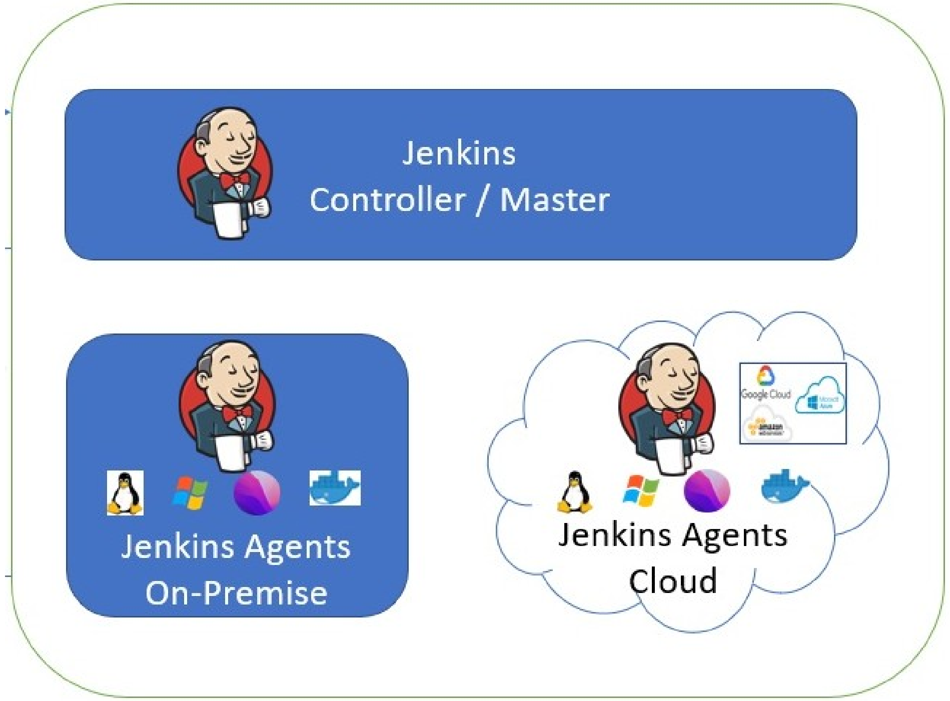
Automatically Provisioned

- Jenkins does not require operators to
  - Create VMs manually
  - Register agents by hand
  - Predict future capacity
- Instead, Jenkins
  - Detects demand
  - Requests new agents dynamically
  - Provisioning logic is configured once and reused automatically



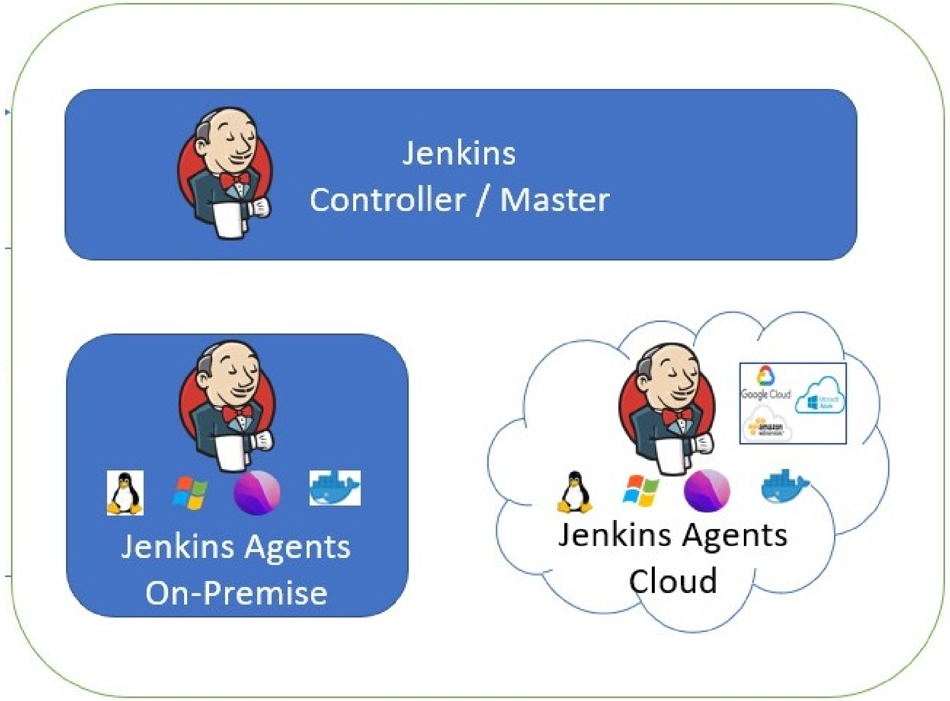
Lifecycle

- Cloud agents
  - Are created when builds queue up
  - Execute one or more builds
  - Shut down when idle
- Eliminates
  - Idle machines
  - Long-running unused agents
  - Manual capacity management



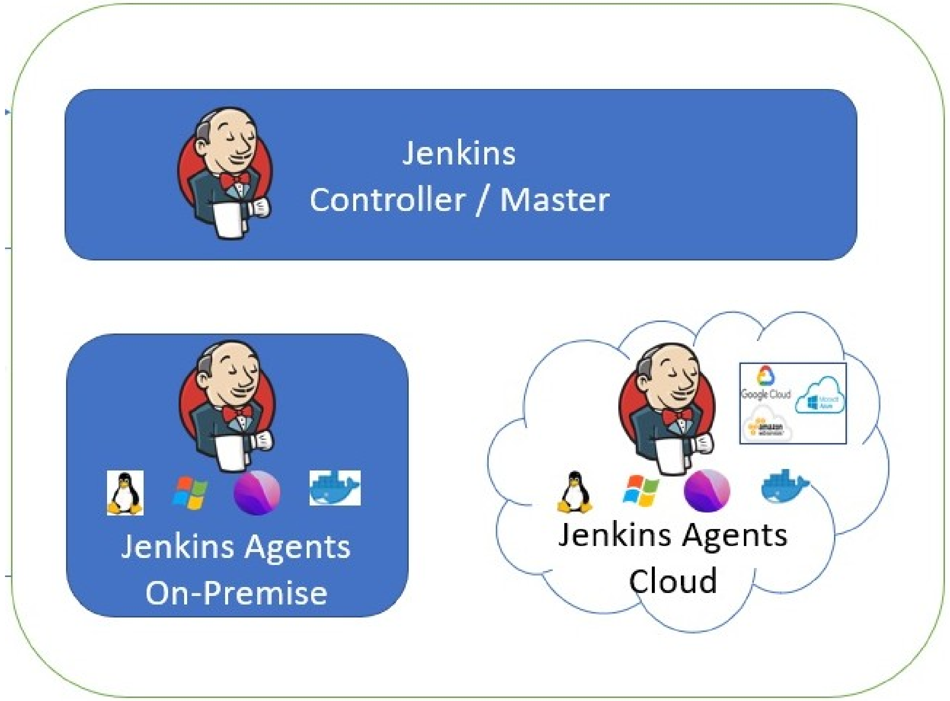
Lifecycle

- Cloud agents are typically terminated
  - After a single build
  - Or after an idle timeout
- Guarantees
  - Clean build environments
  - No workspace contamination
  - No lingering credentials or artifacts



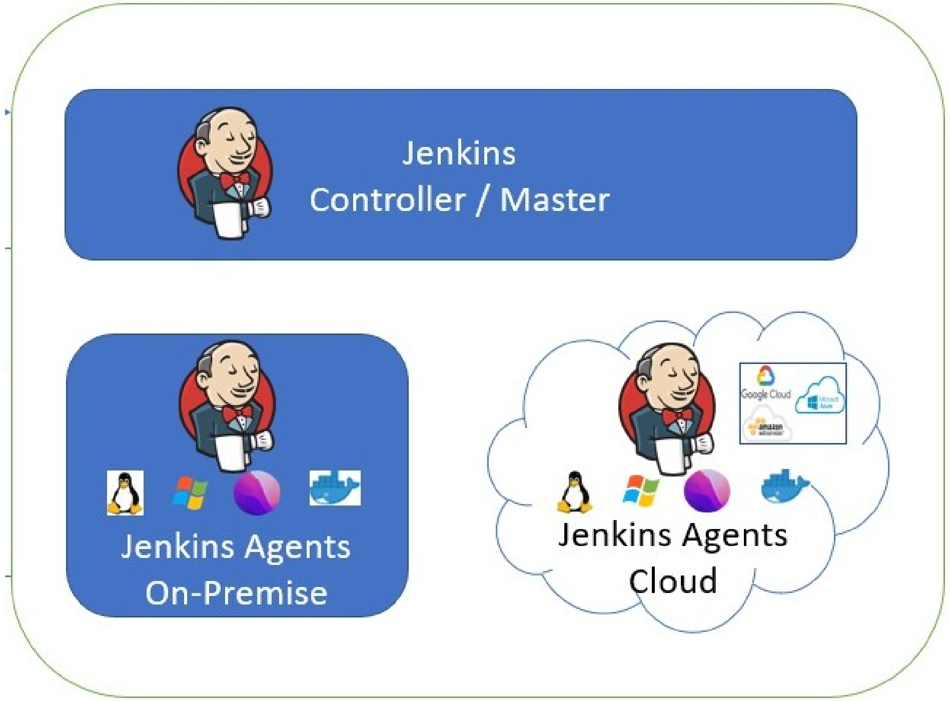
Provisioning Flow

- Typical flow
  - Build queue grows
    - *Jenkins detects that no suitable agent is available*
  - Cloud plugin is triggered
    - *Jenkins evaluates configured cloud templates*
    - *Determines which agent type matches the job*
  - Provisioning request sent
    - *Jenkins calls the cloud provider API*
    - *Requests a new VM instance*
  - VM is created
    - *Image, size, network, and credentials are applied*
    - *Startup scripts install or launch Jenkins agent software*
  - Agent connects to Jenkins
    - *Usually via inbound (JNLP) connection*
    - *Agent registers itself dynamically*
  - Build executes
    - *Pipeline steps run on the new VM*
  - Idle timeout expires
    - *Agent is terminated automatically*
    - *Resources are released back to the cloud*



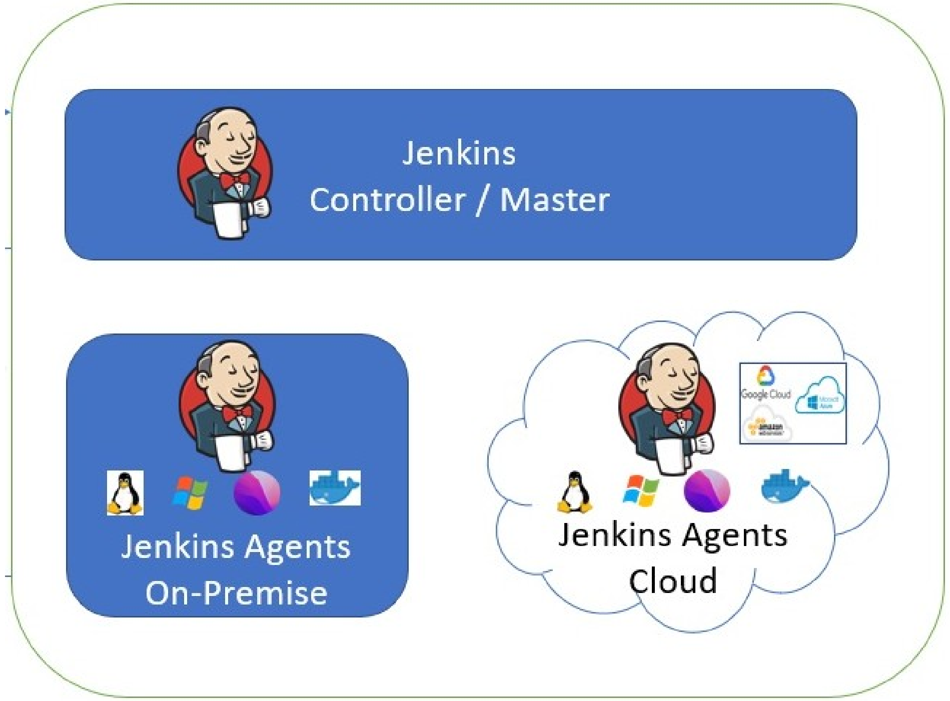
Cloud Provider Integration

- Jenkins integrates natively with
  - AWS EC2
  - Azure Virtual Machines
  - Google Compute Engine (GCE)
  - For each cloud provider, Jenkins can define
    - *VM images (AMIs, images, snapshots)*
    - *Instance sizes*
    - *Network settings*
    - *Labels for job placement*
    - *Maximum instance counts*
    - *Idle termination policies*
  - These definitions act as agent templates



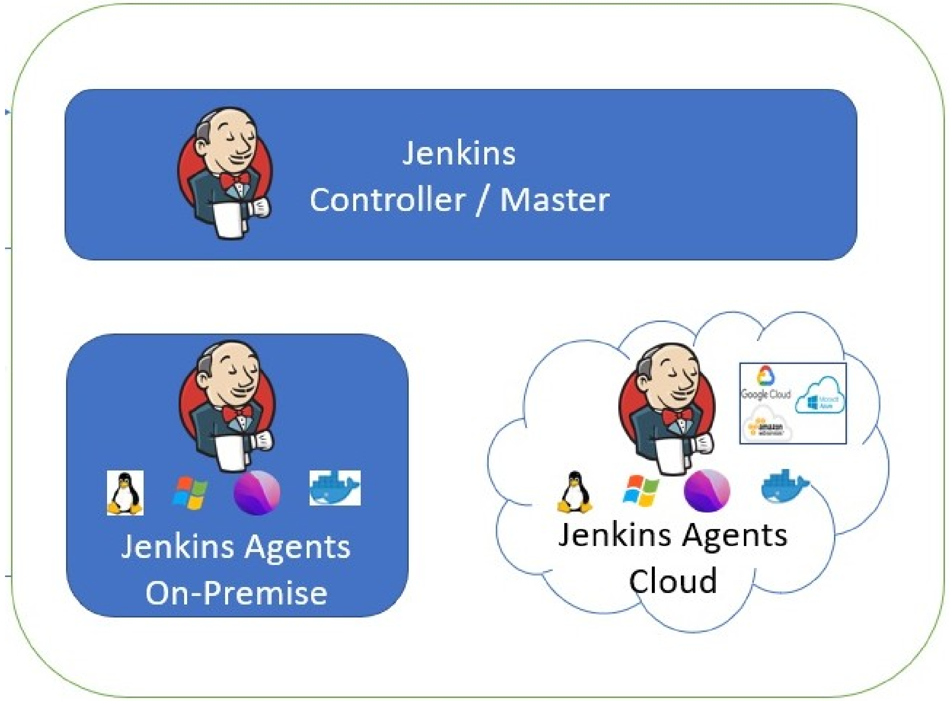
Elastic Scaling

- Cloud agents handle
  - Sudden spikes in commits
  - Parallel pipelines
  - Multiple teams building at once
- Instead of builds waiting in long queues
  - Jenkins requests more agents
- Leads to
  - Faster feedback
  - Higher developer productivity



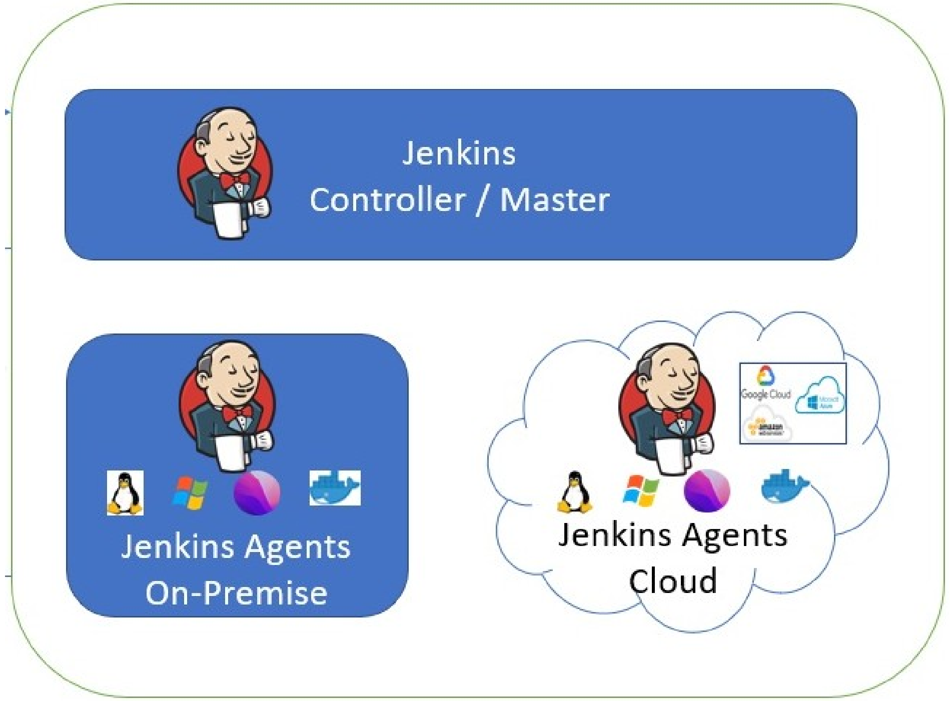
Cost Efficiency

- With cloud agents
  - Costs are pay as you go, not capital investment
  - Idle time is eliminated
  - Resources match actual demand
- Compared to static agents
  - No wasted overnight or weekend capacity from idle agent machines
  - Lower total cost of ownership



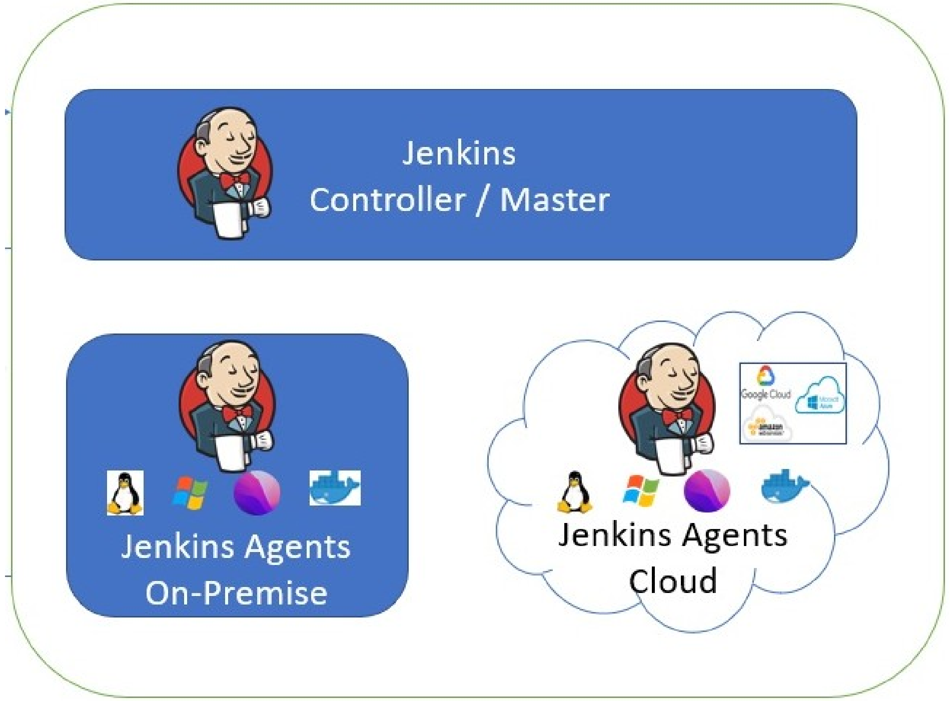
Clean, Reproducible Environments

- Each cloud agent
  - Starts from a known image
  - Has no history from previous builds
  - Eliminates configuration drift
- This improves
  - Build reliability
  - Debugging accuracy
  - Audit trails and governance capabilities



Operational Tradeoffs

- Startup latency
  - Cloud agents:
    - *Take time to provision*
    - *VM startup may take minutes*
  - This makes them
    - *Slower than containers*
    - *Unsuitable for ultra-low-latency CI*
  - Mitigation includes
    - *Warm pools*
    - *Smaller images*
    - *Pre-configured startup scripts*



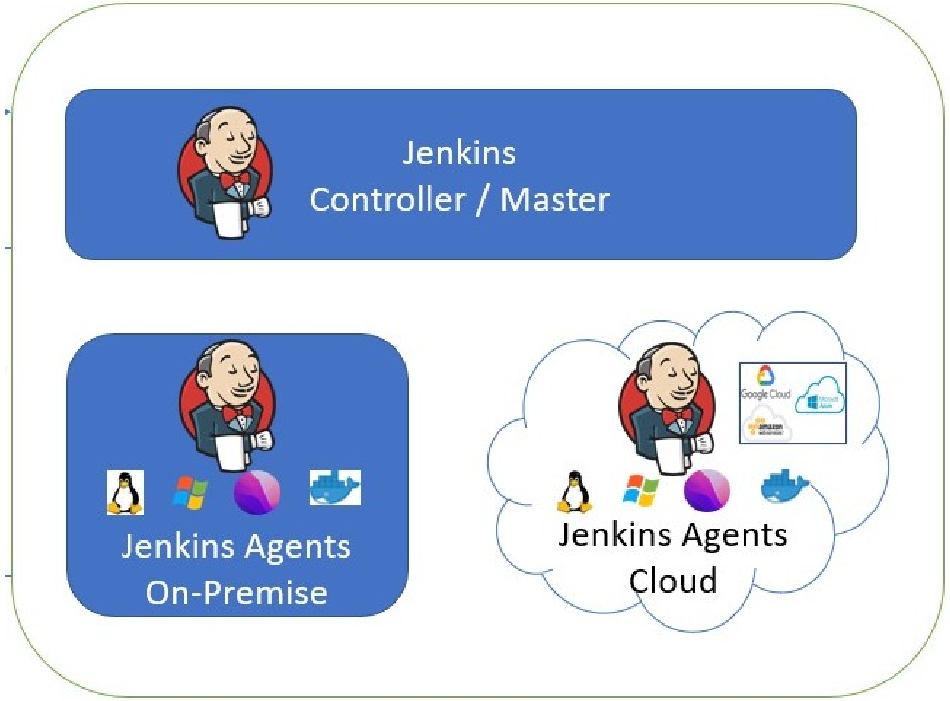
Jenkins Controller / Master

Jenkins Agents On-Premise

Jenkins Agents Cloud

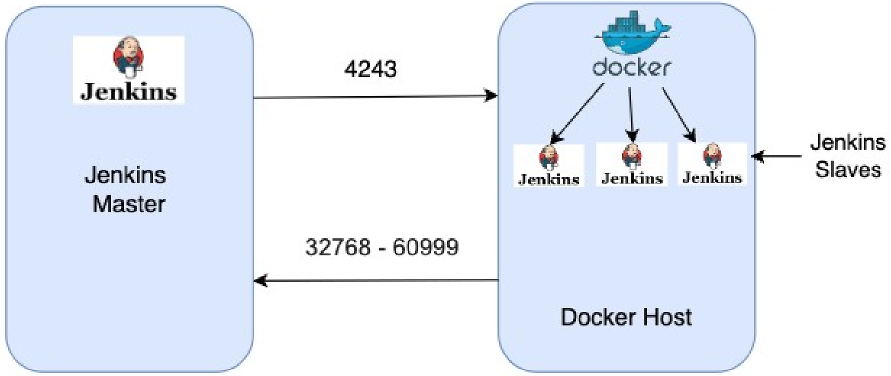
Operational Tradeoffs

- Cloud dependency
  - Cloud agents require:
    - *Cloud availability*
    - *API access*
    - *IAM permissions*
  - Failures can occur if
    - *API quotas are exceeded*
    - *Credentials expire*
    - *Regions are unavailable*



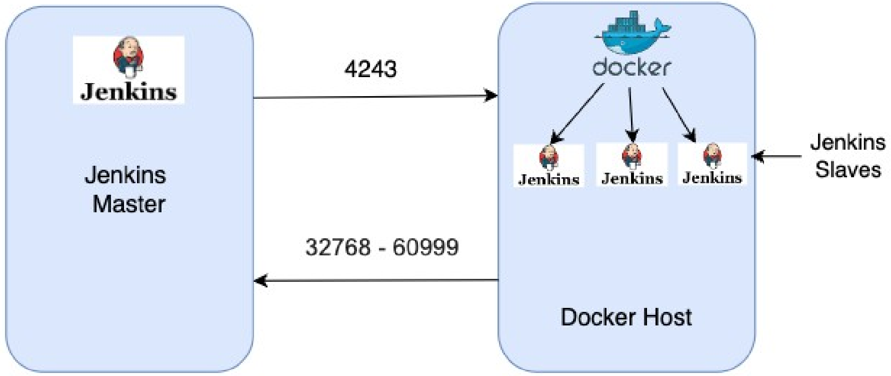
Docker Agents

- A Docker agent
  - Runs inside a Docker container
  - Executes pipeline steps within that container
  - Exists only for the duration of the build
    - *Ephemeral by default*
  - Docker is not used here primarily to build images
  - But provides a controlled, repeatable runtime environment for Jenkins pipelines



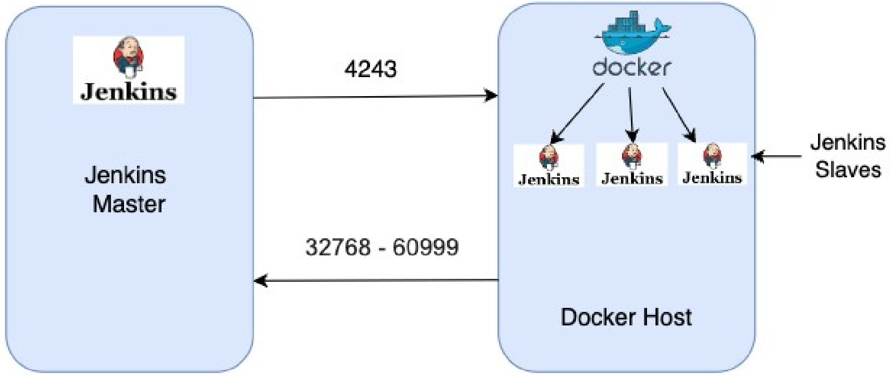
Docker Agents Benefits

- Tool version conflicts can be resolved
  - On static agents
    - *One job needs Java 11*
    - *Another needs Java 17*
    - *Another needs Node 18*
  - This leads to
    - *PATH manipulation*
    - *Fragile setups*
    - *Frequent breakage*
  - Docker agents solve this by
    - *Encapsulating tools into images*
    - *Running each build in its own environment*



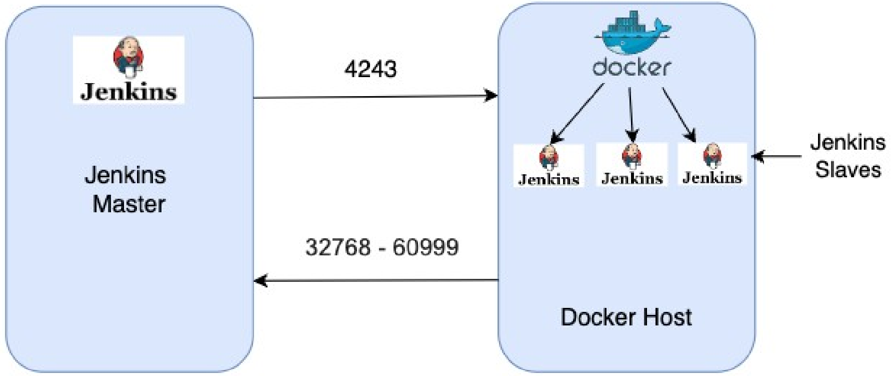
Docker Agents Benefits

- No dirty workspaces
  - Long-lived agents accumulate
    - *Temporary files*
    - *Cached dependencies*
    - *Leftover artifacts*
    - *Broken state from failed builds*
  - Docker agents
    - *Start from a clean file system*
    - *End with container deletion*
    - *No cleanup scripts required.*



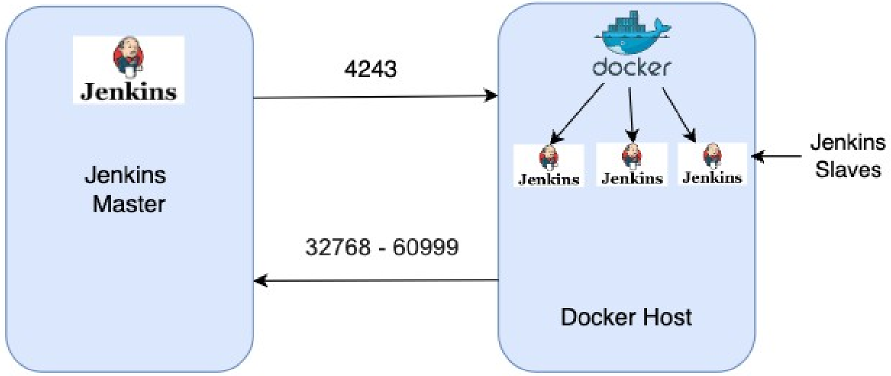
Docker Agents Benefits

- Fast provisioning compared to VMs
  - Docker containers
    - *Start in seconds*
    - *Are lighter than virtual machines*
    - *Share the host kernel*
  - This makes Docker agents
    - *Faster than cloud VM agents*
    - *Ideal for short-lived CI tasks*

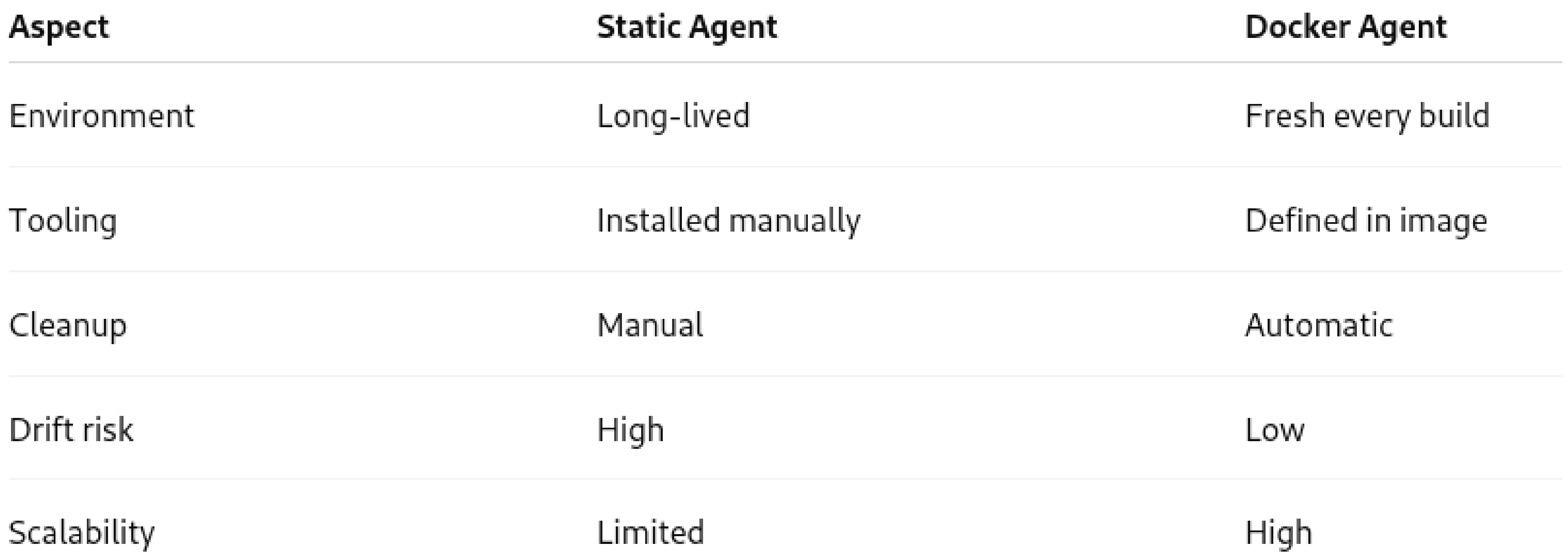


Docker Agent Lifecycle

- Control flow
  - Pipeline requests a Docker agent
    - *Based on Jenkinsfile definition*
  - Jenkins pulls the container image
    - *From a registry (Docker Hub, ECR, etc.)*
    - *Cached images start faster*
  - Container is started
    - *Workspace is mounted*
    - *Environment variables are injected*
    - *Credentials are provided securely*
  - Pipeline steps execute
    - *sh, build tools, tests run inside container*
  - Container stops
    - *Workspace may persist (depending on configuration)*
    - *Container is destroyed*



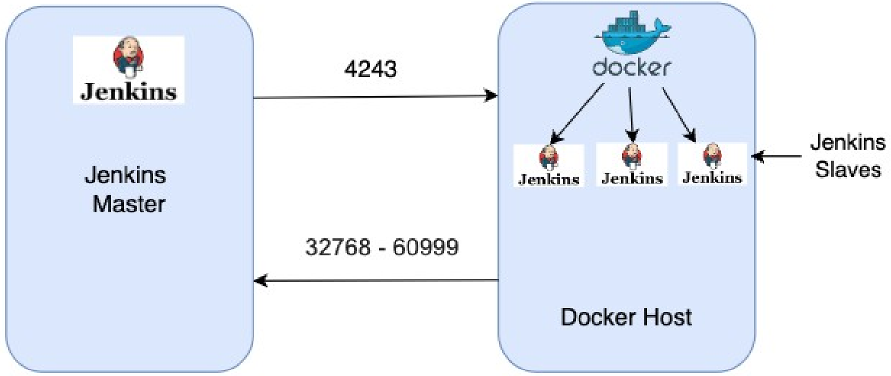
Docker Agents vs Static Agents



Aspect	Static Agent	Docker Agent
Environment	Long-lived	Fresh every build
Tooling	Installed manually	Defined in image
Cleanup	Manual	Automatic
Drift risk	High	Low
Scalability	Limited	High

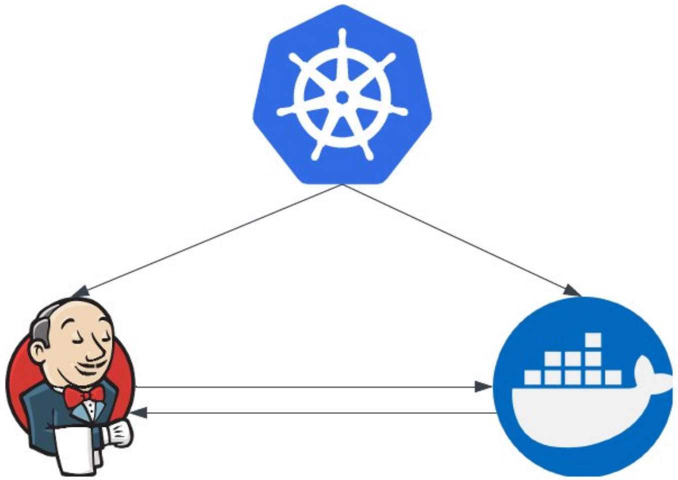
Limitations and Tradeoffs

- Requires Docker infrastructure
  - Docker agents need
    - *Docker installed*
    - *Proper daemon access*
    - *Secure host configuration*
- Not fully isolated like VMs
  - Containers
    - *Share the host kernel*
    - *Are not full virtual machines*
  - For high-security workloads
    - *Additional hardening may be required*
    - *Kubernetes or VM isolation may be preferred*



Kubernetes Agents

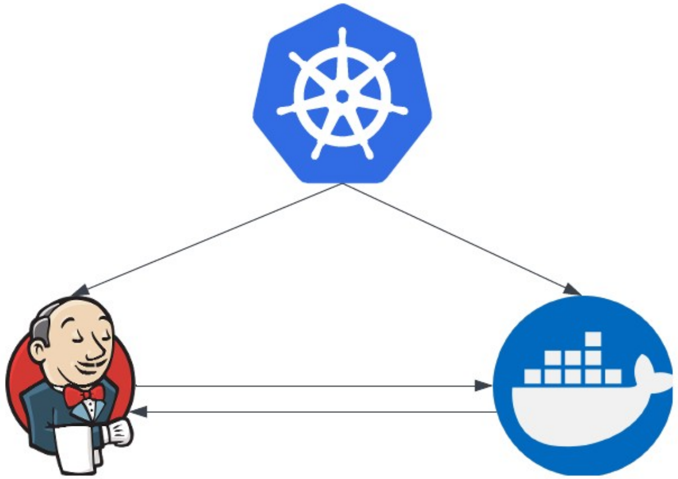
- Jenkins uses Kubernetes pods as build agents
  - Instead of
    - *Long-lived machines*
    - *Pre-registered agents*
  - Jenkins dynamically creates short-lived pods to execute builds
  - Each pod acts as a Jenkins agent
  - Exists only for the duration of the build
  - Is destroyed afterward



Agent as a Pod

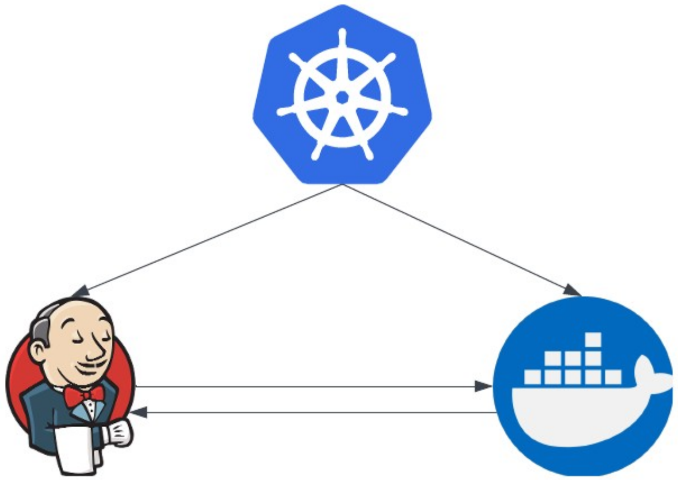
- In this model:
  - One Jenkins agent = one Kubernetes pod

    The pod contains
    - *A Jenkins agent container*
    - *One or more tool containers*
  - The pod has
    - *Its own resources and memory limits*
    - *Its own filesystem*
    - *Its own lifecycle*
  - This eliminates
    - *Agent reuse*
    - *Configuration drift*
    - *Manual cleanup*



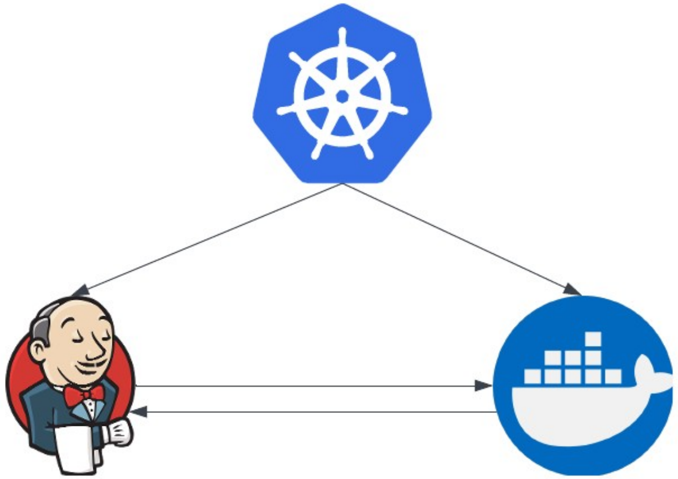
Kubernetes Benefits

- Automatic scaling
  - Kubernetes
    - *Schedules pods automatically*
    - *Scales with demand*
    - *Manages node capacity*
  - Jenkins benefits because
    - *Agents appear only when builds need them*
    - *No fixed upper limit (within cluster capacity)*
    - *CI scales naturally with developer activity*



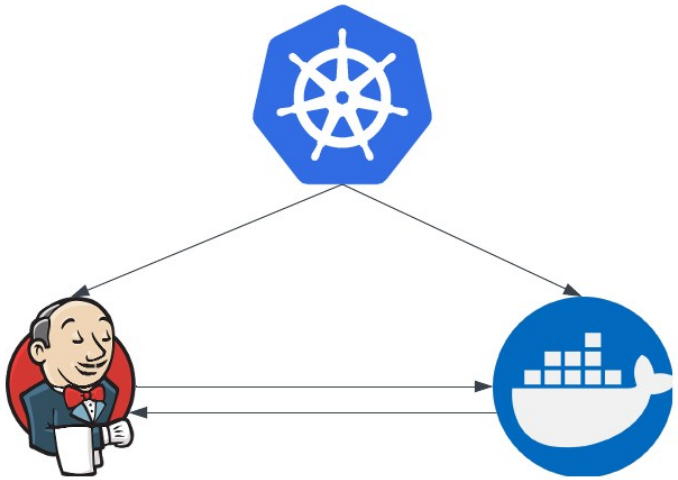
Kubernetes Benefits

- Resource isolation
  - Each Kubernetes pod
    - *Has defined CPU and memory limits*
    - *Cannot consume resources beyond its allocation*
    - *Is isolated from other builds*
  - This prevents
    - *One build starving others*
    - *Unpredictable performance*
    - *Noisy-neighbor problems*



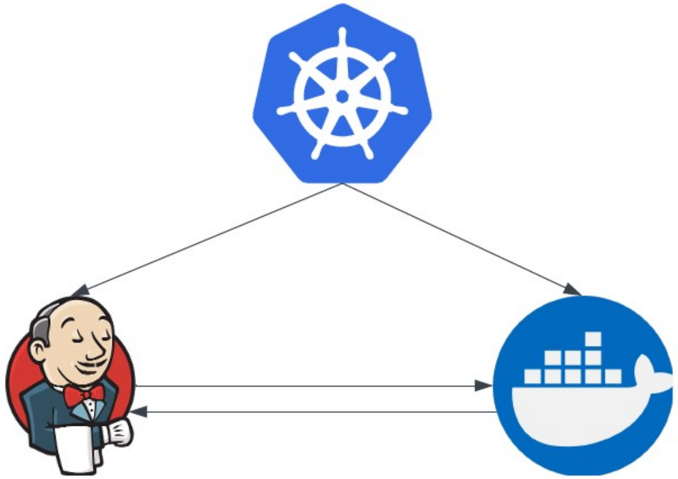
Kubernetes Benefits

- Self-healing
  - Kubernetes
    - *Restarts failed pods*
    - *Reschedules pods if nodes fail*
    - *Handles infrastructure failures transparently*
  - For Jenkins
    - *Failed agent nodes don’t take down CI*
    - *Builds fail fast instead of hanging*
    - *Platform resilience improves dramatically*



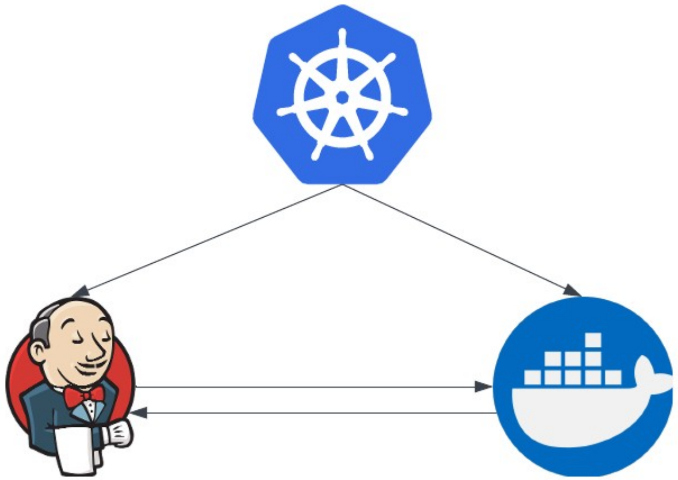
Kubernetes Agent Lifecycle

- Agent flow
  - Pipeline starts
    - *Jenkins evaluates agent requirements*
  - Jenkins requests an agent
    - *Via the Kubernetes plugin*
    - *Defines pod template (containers, resources)*
  - Kubernetes schedules the pod
    - *Selects a node*
    - *Pulls required images*
  - Agent container connects to Jenkins
    - *Usually via inbound (JNLP) connection*
    - *Registers dynamically*
  - Build executes inside the pod
    - *Steps run in one or more containers*
    - *Workspace is shared*
  - Build completes
    - *Pod is terminated*
    - *All resources are released*



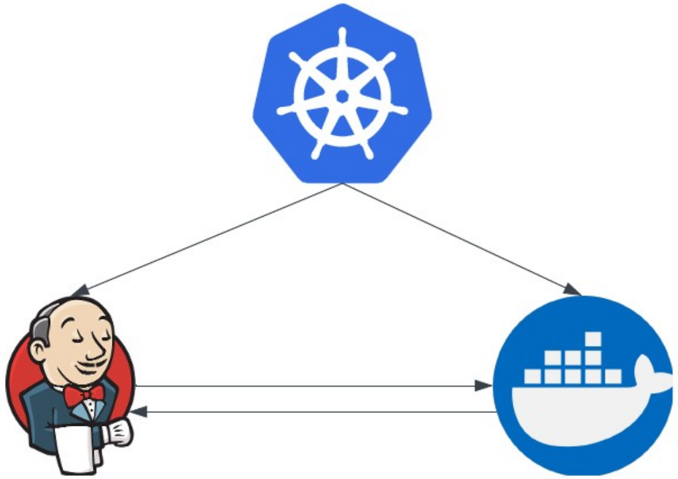
Multi-Container Pods

- A Kubernetes pod can contain
  - Multiple containers running side-by-side
  - Same pod containers share
    - *Filesystem (workspace)*
    - *Network*
    - *Lifecycle*
- Instead of
  - Installing all tools in one Docker image
  - Creating massive “everything” images
- Jenkins can
  - Use small, purpose-built containers
  - Combine them at runtime in a single pod

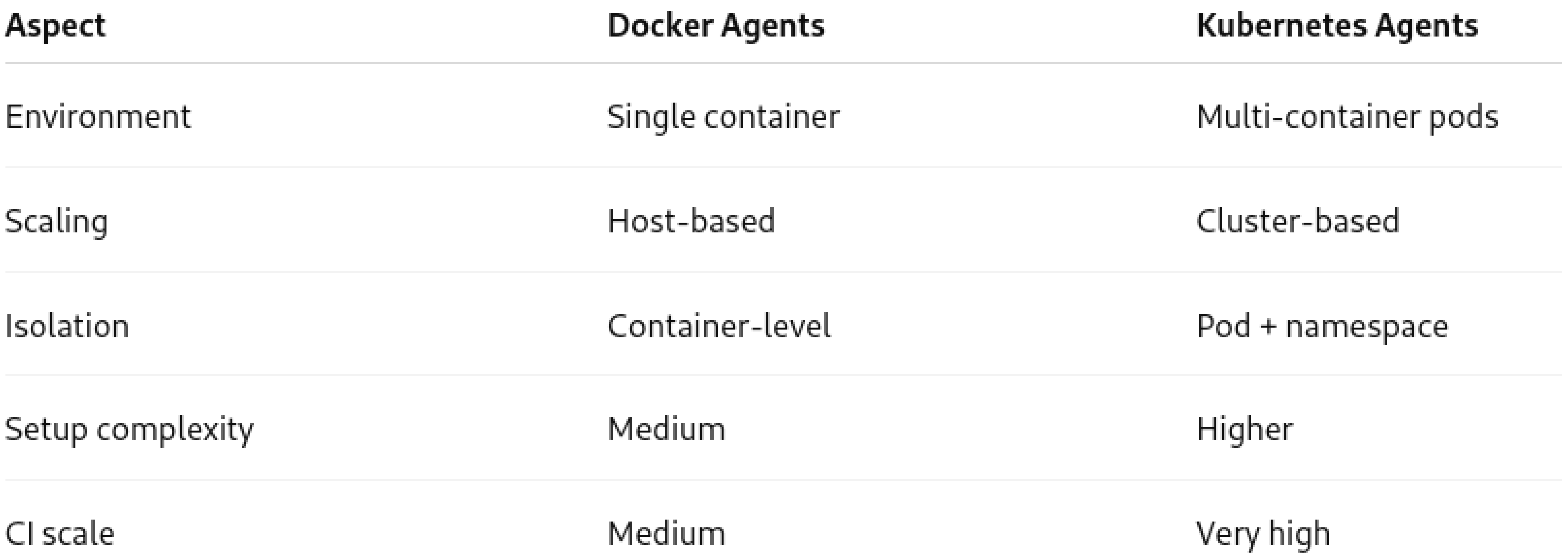


Multi-Container Patterns

- Build + Test
  - Container 1: build tools (e.g. Maven)
  - Container 2: test runtime
- Build + Security Scan
  - Container 1: compiler/build
  - Container 2: security scanner
  - Container 3: reporting tool
- Build + Docker
  - Container 1: build environment
  - Container 2: Docker daemon (sidecar)
- All containers
  - Share the same workspace
  - Are destroyed together



Comparison



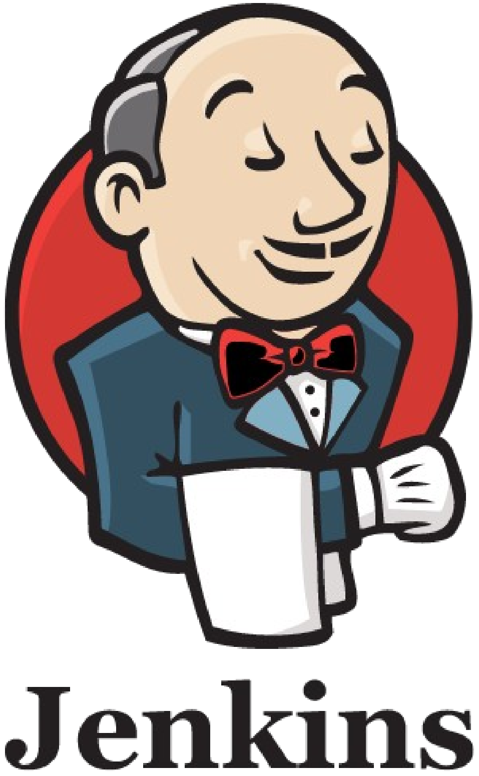
Aspect	Docker Agents	Kubernetes Agents
Environment	Single container	Multi-container pods
Scaling	Host-based	Cluster-based
Isolation	Container-level	Pod + namespace
Setup complexity	Medium	Higher
CI scale	Medium	Very high

Jenkins Workload

- Jenkins is a scheduler
  - Its primary job is to
    - *Accept work (builds)*
    - *Decide where that work should run*
    - *Ensure work runs safely and efficiently*
- The build queue
  - When a job is triggered
    - *It does not immediately start*
    - *It is placed into the Jenkins build queue*
  - A job stays in the queue until
    - *A suitable agent is available*
    - *An executor on that agent is free*

Jenkins Workload

- Why jobs stay queued
  - No agents are available
  - Agents exist but lack required labels
  - All executors are busy
  - Resource limits are reached
- The queue contains diagnostic information explaining why a job is waiting

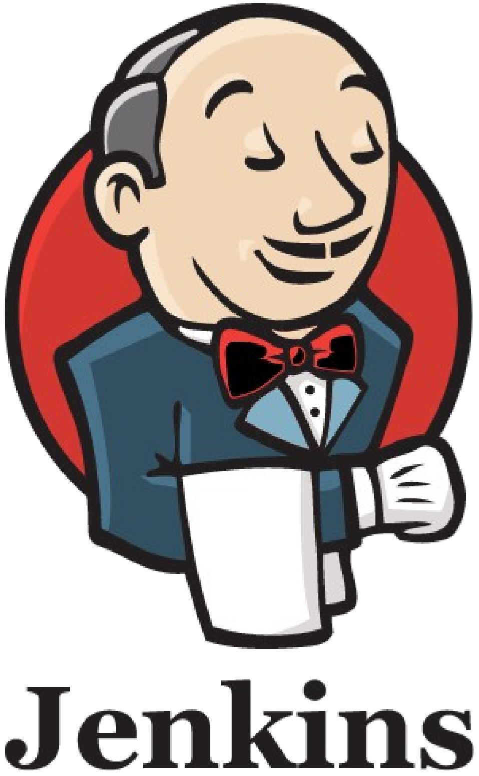


Executors

- An executor is a logical slot for running a build
  - Executors are not CPUs
  - Executors are not threads
  - They represent concurrent build capacity
    - *A combination of factors like thread and CPU capability*
- Too many executors on one agent
  - Causes CPU contention
  - Increases memory pressure
  - Slows all builds
  - Makes failures harder to diagnose

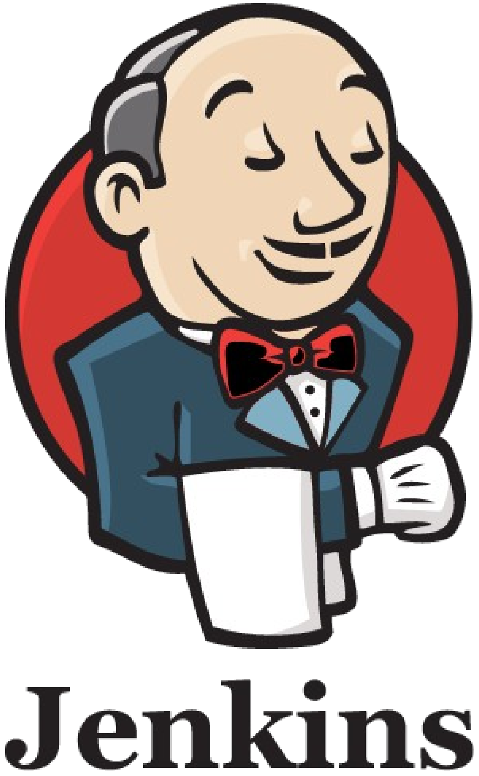
Performance Considerations

- Number of executors
  - Key tradeoff
    - *Too few: idle agents, long queues*
    - *Too many: resource contention*
  - Rule of thumb
    - *Prefer one executor per agent to avoid resource contention*
    - *Scale with more agents, not more executors*
- Agent availability
  - If agents are
    - *Offline*
    - *Misconfigured*
    - *Slow to start*
  - Then
    - *Queue grows*
    - *Feedback slows*
    - *Developer productivity drops*
  - This is why ephemeral and cloud agents are so powerful



Performance Considerations

- Build Duration
  - Long-running builds
    - *Block executors*
    - *Reduce throughput*
    - *Increase queue time*
  - Common causes
    - *Monolithic pipelines*
    - *Unbounded test suites*
    - *Slow external dependencies*
  - Mitigations
    - *Break pipelines into stages*
    - *Parallelized work*
    - *Split tests*



Performance Considerations

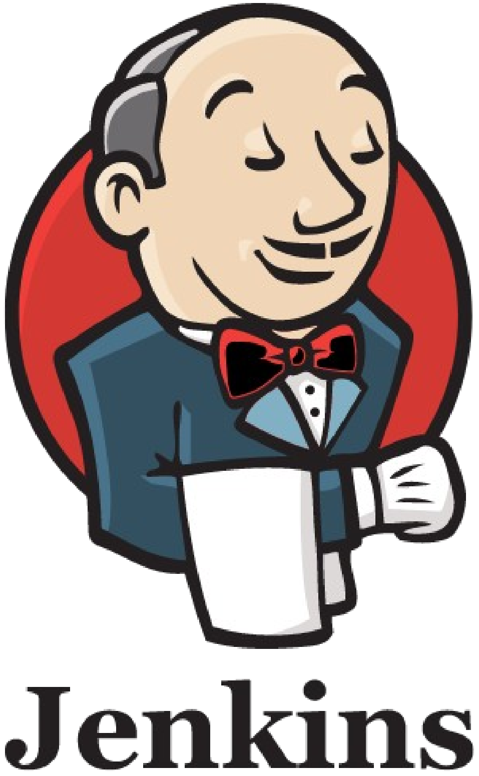
- Resource Limits
  - Agents must have
    - *Enough CPU*
    - *Enough memory*
    - *Adequate disk*
  - If resources are under-sized
    - *Builds slow down*
    - *Fail intermittently*
    - *Cause noisy-neighbor effects*
  - If over-sized
    - *Resources are wasted*

Best Practices for Performance

- Minimize executors on the controller
  - Set controller executors to zero
    - *Jenkins controller does not use an executor to run its scheduling tasks*
    - *Prevents accidental workload execution*
    - *Protects Jenkins stability*

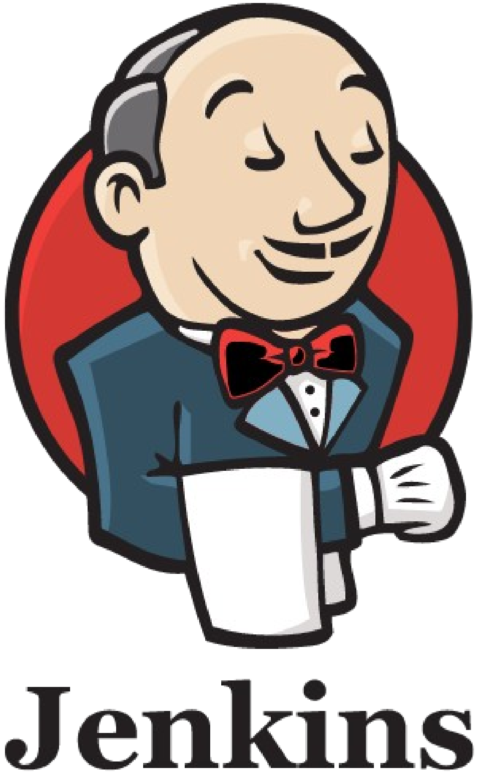
Best Practices for Performance

- Prefer ephemeral agents
  - Ephemeral agents
    - *Start clean*
    - *Avoid drift*
    - *Scale automatically*
    - *Fail fast and visibly*
  - This improves
    - *Reliability*
    - *Throughput*
    - *Security*



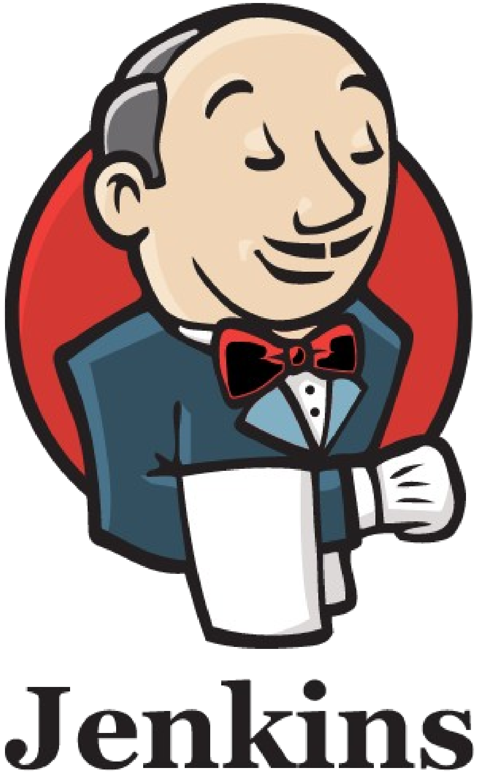
Best Practices for Performance

- Right-size agent resources
  - Avoid
    - *One “mega-agent” doing everything*
  - Use smaller, purpose-built separate agents for
    - *Build*
    - *Test*
    - *Scan*
- Avoid long-running builds
  - Best practices
    - *Enforce timeouts*
    - *Fail fast*
    - *Split pipelines*
    - *Parallelized aggressively*



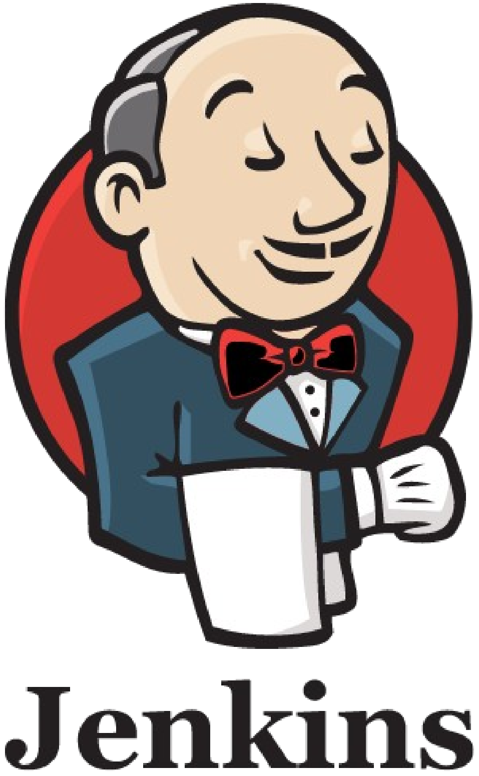
Scaling Strategies

- Add more agents (horizontal scaling)
- Use ephemeral or cloud agents
- Parallelized pipelines
- Common performance anti-patterns
  - Running builds on the controller
  - Increasing executors instead of agents
  - Overloading single agents
  - Ignoring queue metrics
  - Treating Jenkins like a single server



Performance Tips

- To manage performance, teams should monitor
  - Queue length
  - Queue wait time
  - Agent utilization
  - Build duration trends
  - Failure rates
- These metrics guide
  - Capacity planning
  - Agent sizing
  - Pipeline optimization



Questions