How to become a Staff AI Engineer

Chapter 5: Operating Systems, Linux, and Containers for AI Engineering

Celso Sousa
LinkedIn: https://www.linkedin.com/in/celso-sousa/?locale=en-US
Portfolio: https://celsosousa.com.br/project-portfolio/
Ebook: https://celsosousa.com.br/ebook-staff-ai-engineer/

Chapter Objective

AI systems do not execute in the abstract. Every model, agent, pipeline, retrieval service, or API competes for CPU, memory, disk, network, and accelerators within infrastructure whose behavior is governed by the operating system.

For a Staff AI Engineer, understanding operating systems does not mean knowing every internal detail of a kernel. The objective is to develop a mental model capable of explaining why a service exhibits high latency, why a container was terminated, why increasing the number of workers reduces throughput, why a GPU remains idle, or why an apparently healthy workload enters an Out-of-Memory condition when exposed to real traffic.

This knowledge becomes particularly relevant in environments that combine high concurrency, inference, parallel processing, and operational isolation:

- **Highly concurrent APIs compete for shared resources:** More requests increase pressure on CPU, memory, sockets, and file descriptors;
- **Inference workloads have distinct resource profiles:** Models may require large amounts of RAM or VRAM and exhibit utilization patterns very different from conventional web applications;
- **Data pipelines may be CPU- or I/O-bound:** Parsing, serialization, file reads, and data transfers can become bottlenecks independently of the model;
- **Agents may execute tools and external code:** This introduces additional requirements for isolation, resource limiting, and security;
- **Parallel processing has overhead:** Increasing the number of processes or threads does not guarantee higher throughput;
- **Containers introduce operational boundaries:** However, they still share host resources and the host kernel;
- **Kubernetes depends on operating system mechanisms:** Scheduling, resource limits, and isolation remain grounded in processes, namespaces, and cgroups;
- **GPUs introduce their own constraints:** Accelerator memory, model loading, batching, and data feeding must be considered;
- **Long-running services accumulate operational state:** Leaks, unbounded caches, and resource growth may emerge only after hours or days of execution.

These scenarios require operating system concepts to be analyzed from a production perspective rather than exclusively from an academic one. Several principles will recur throughout this chapter:

- **Processes define execution boundaries:** They affect isolation, parallelism, memory, communication, and fault tolerance;
- **Threads enable concurrency within a process:** They reduce certain communication costs but introduce shared state and synchronization problems;
- **Memory is a finite and critical resource:** OOM conditions, leaks, paging, and unbounded caches can bring down applications even when CPU remains available;
- **Linux provides the operational foundation for modern AI infrastructure:** Processes, sockets, filesystems, permissions, and resource limits are fundamental to troubleshooting;
- **Containers provide operational isolation, not complete virtual machines:** This distinction affects security, performance, and capacity planning;
- **Namespaces and cgroups underpin modern containers:** They explain much of the behavior observed in Docker and Kubernetes;
- **Resource utilization must be connected to economics:** Idle CPU, RAM, VRAM, and hardware represent real product cost.

The ultimate objective is to develop the ability to answer a fundamental Staff Engineering question: **what is happening between the application code and the physical resources that actually execute that code?**

5.1. Processes, Threads, and Concurrency

A process represents an instance of a program in execution. It owns state, memory, and resources managed by the operating system.

In AI Engineering, processes appear in web servers, workers, jobs, pipelines, inference services, and tools executed by agents. Understanding their boundaries is important because seemingly simple decisions, such as increasing the number of workers, can dramatically change CPU utilization, memory consumption, latency, and reliability.

5.1.1. Processes vs. Threads

Processes and threads are different mechanisms for executing concurrent work. A process has its own address space. Threads belong to a process and share much of that state.

This difference creates important trade-offs:

- **Processes provide stronger isolation:** A crash or memory failure tends to remain confined to that execution unit;
- **Threads share memory with lower overhead:** Communication between tasks can be efficient because structures do not necessarily need to be copied;
- **Shared memory increases concurrency risk:** Race conditions, improper synchronization, and state corruption become possible;
- **Processes have higher operational cost:** Creating processes and moving data between them generally costs more than using threads;
- **Threads work well for certain I/O-bound workloads:** When much of the execution time is spent waiting for network or storage, concurrency can increase system utilization;
- **Processes are particularly useful for isolation and CPU-bound workloads:** They allow computation to be distributed across multiple cores while establishing clearer failure boundaries.

Consider a pipeline responsible for computationally intensive document parsing. If parsing continuously consumes CPU, distributing work across processes can increase parallelism.

An agent making multiple calls to external APIs has a different profile. In that case, much of the execution time may be spent waiting for I/O, making threads or asynchronous mechanisms more appropriate alternatives.

The number of tasks alone therefore does not determine the concurrency strategy. What matters is understanding where the workload spends its time and what degree of isolation is required.

5.1.2. Scheduling, Workers, and Effective Capacity

The operating system must decide which processes and threads receive CPU time at any given moment. This mechanism is known as scheduling. In shared infrastructure, being in a running state does not mean having continuous access to the CPU. Different workloads compete for the same cores, and the scheduler must distribute capacity among them.

This dynamic has important implications for AI systems:

- **High CPU utilization can increase tail latency:** The service may remain available while requests spend more time waiting for CPU;
- **Adding workers can eventually reduce performance:** Beyond a certain point, additional processes increase contention rather than throughput;
- **Batch workloads can interfere with interactive workloads:** A computationally intensive pipeline can degrade a low-latency API when both compete for the same resources;
- **Requests and limits influence container behavior:** Declared resources and the resources actually available to the workload must be aligned;
- **Oversubscription can increase efficiency:** When workloads do not simultaneously consume all requested capacity, sharing resources improves utilization;
- **Oversubscription also increases risk:** During simultaneous peaks, CPU contention can cause significant degradation.

A recurring mistake is assuming that throughput grows proportionally with the number of workers. If a machine has only a few available cores and dozens of CPU-bound processes, increasing worker count creates more entities competing for the same physical capacity. The system may spend an increasing fraction of its time coordinating execution rather than performing useful work.

At the Staff level, the relevant question is not merely how many workers exist, but what **effective capacity** is actually available to the system.

5.1.3. Context Switching and Coordination Cost

Context switching occurs when the operating system suspends one task and starts executing another. Part of the execution state must be preserved and later restored.

Context switches are natural in multitasking systems. The problem emerges when the architecture introduces enough concurrency that coordination becomes a significant portion of total cost:

- **Too many threads can increase context switching:** The CPU spends more time alternating among execution units;
- **Too many processes also introduce overhead:** Stronger isolation does not eliminate scheduling cost;

- **Excessively small tasks can be inefficient:** The cost of distributing, scheduling, and collecting results may exceed the cost of the work itself;
- **Lock contention wastes capacity:** Threads may be scheduled only to discover that they must wait for another thread to release a resource;
- **Parallelism does not eliminate physical limits:** Core count, available memory, and bandwidth remain finite.

This leads to an important principle: **parallelism has overhead.** Dividing a problem into thousands of units does not automatically produce higher throughput. Each unit must perform enough useful work to justify the cost of scheduling, coordination, and data movement.

In AI pipelines, batching frequently reduces this overhead:

- **Batching reduces scheduling decisions:** More useful work can be performed per execution unit;
- **Batching reduces network calls:** Operations can be grouped into fewer requests;
- **Batching can reduce inference cost:** Infrastructure can process multiple inputs together;
- **Batching reduces part of the coordination cost:** Fewer independent units need to be managed.

However, batching also increases memory consumption and can introduce waiting latency. Optimization must consider the complete system.

5.1.4. Inter-Process Communication and Data Movement

Isolated processes frequently need to exchange data. Because their memory spaces are separate, this communication requires explicit mechanisms. Common alternatives include pipes, sockets, shared memory, queues, files, and message brokers.

The choice should consider communication volume, frequency, durability, and topology:

- **Pipes and queues simplify local communication:** They are appropriate for certain producer-consumer patterns between related processes;
- **Shared memory reduces copying:** It can be useful when data volumes are large but requires careful control of shared access;
- **Sockets enable generic communication:** They can connect local processes or processes running on different machines;
- **Message brokers increase decoupling and durability:** They become more appropriate when communication is part of a distributed system;
- **Files can serve as an intermediate mechanism:** However, they introduce filesystem semantics and latency.

In AI workloads, data movement can consume a significant fraction of total execution time.

Consider multiple workers processing large images. If every input must repeatedly be serialized, copied, and deserialized, communication cost can eliminate much of the benefit obtained from parallelism. The same problem occurs when large arrays, embeddings, or intermediate results constantly cross process boundaries. The architectural principle is simple: **whenever possible, keep compute close to the data.**

The Staff AI Engineer must evaluate not only how much processing has been parallelized, but also how much additional work was created to move the data required by that processing.

5.2. Memory, Locality, and Capacity Planning

Memory is one of the most critical resources in AI Engineering. Model weights, embeddings, caches, contexts, batches, and intermediate representations can consume large amounts of RAM or VRAM. Unlike high CPU utilization, which often produces gradual latency degradation, extreme memory pressure can terminate processes abruptly. For this reason, memory behavior must be treated as part of the architecture rather than merely as an infrastructure metric.

5.2.1. Virtual Memory

Virtual memory gives each process the abstraction of owning its own address space. The operating system maps virtual addresses to physical memory and manages that mapping throughout execution.

This layer provides both isolation and flexibility:

- **Processes have independent address spaces:** This makes arbitrary access to another process's memory more difficult;
- **Virtual memory can be reserved without being fully resident in RAM:** Address reservation and physical memory consumption are not equivalent;
- **Pages can be shared in a controlled manner:** Libraries and other regions can be reused;
- **Virtual addressing simplifies memory management:** Applications do not need to know the physical location of their data;
- **Virtual memory contributes to isolation:** It participates in the fundamental mechanisms that protect processes from one another.

This abstraction also explains why different memory metrics can appear contradictory. During troubleshooting, at least three concepts must be distinguished:

- **Virtual memory represents reserved address space:** A high value does not necessarily imply equivalent RAM consumption;
- **Resident memory represents pages currently present in physical memory:** This metric is more directly related to actual RAM pressure;
- **Shared memory can be used by multiple processes:** Naively summing memory consumption reported by individual processes can overestimate physical utilization.

Without understanding these distinctions, memory pressure diagnostics can lead to incorrect decisions.

5.2.2. Paging and Data Locality

Paging divides memory into blocks called pages. The operating system maps and moves these pages according to workload requirements.

When data required for execution is not available where expected, page faults and additional operations may be required to make it accessible.

This behavior directly affects performance:

- **Page faults add latency:** Data must become available before execution can continue;
- **High memory pressure increases operating system work:** The application may degrade before a definitive failure occurs;
- **Random access to large structures reduces efficiency:** The access pattern may use caches and memory inefficiently;

- **Locality improves hardware utilization:** Data accessed close together in time or space tends to make better use of memory hierarchies;
- **Memory layout can affect throughput:** This effect becomes relevant in numerical workloads and vector search.

This point connects algorithmic decisions to the physical architecture of the machine. Two structures capable of representing the same information can exhibit very different performance characteristics if one provides good locality while the other scatters accesses across large memory regions.

At scale, efficiency depends not only on asymptotic complexity. It also depends on how the actual access pattern interacts with caches, pages, and physical memory.

5.2.3. Memory Mapping

Memory mapping allows files or storage regions to be associated with a process's address space. The application can access portions of those files without necessarily performing an explicit full copy into an intermediate structure.

This mechanism is particularly useful for large artifacts:

- **Large datasets can be accessed progressively:** The entire content does not need to be loaded before processing begins;
- **Indexes can exploit on-demand access:** Only regions actually used need to be materialized;
- **Binary files can be treated similarly to memory:** This simplifies certain access patterns;
- **Model parameters can be accessed efficiently:** Depending on the format and runtime, not all content needs to be handled in the same way at the same time.

The main benefits derive from interaction with the operating system:

- **Lazy loading reduces upfront work:** Pages are accessed as needed;
- **Fewer copies can reduce overhead:** The operating system participates directly in management;
- **Pages can be shared across processes:** This can reduce duplication in certain scenarios;
- **Large files become easier to address:** The application operates over a memory abstraction.

However, memory mapping does not eliminate hardware limits. If a workload accesses nearly every region of a file using a highly random pattern, the system may still produce intensive I/O and poor locality. Working with a file larger than available RAM does not make RAM capacity irrelevant. Efficiency continues to be determined by the working set and the access pattern.

5.2.4. OOM and Peak Memory

Out-of-Memory occurs when the system cannot satisfy a memory requirement. On Linux, extreme pressure may trigger the OOM killer. In containers, a similar condition can occur even when the host still has available RAM because the workload has reached its specific memory limit.

AI workloads are particularly exposed to this problem:

- **Large models structurally consume memory:** Simply loading model weights can use a significant portion of capacity;
- **Larger batches increase peak memory:** Throughput improvements may come at the cost of higher memory consumption;

- **Unbounded caches grow continuously:** A latency optimization can become a cause of unavailability;
- **Memory leaks accumulate state over time:** Short tests may not expose the problem;
- **Multiple workers may duplicate data:** Each process may load its own copy of models or structures;
- **Very large inputs can create unpredictable spikes:** Arbitrary documents or payloads increase operational risk;
- **Higher concurrency increases aggregate memory:** A service stable under one request may fail under real traffic.

For this reason, capacity planning cannot rely exclusively on average consumption. The workload must be analyzed against expected peak memory.

Several controls reduce risk:

- **Caches need explicit limits:** No long-running cache should depend solely on available memory;
- **Batch size must be treated as a resource parameter:** Increasing it may improve throughput until memory becomes the bottleneck;
- **Inputs must have limits:** Users should not be able to induce arbitrary memory allocations;
- **Memory growth must be observed over time:** Monotonic growth is different from high but stable utilization;
- **Worker count must account for RAM per process:** In many systems, memory constrains parallelism before CPU does;
- **Capacity planning must account for startup and runtime peaks:** Model initialization may have a different profile from steady state.

Resource limits also serve a security function. A system without limits may be vulnerable to attacks or accidental resource exhaustion.

5.3. Linux as the Operational Layer of AI Engineering

Linux underpins much of modern cloud infrastructure, containers, Kubernetes, and inference servers. Operating at Staff level does not require mastering every internal kernel subsystem. However, it does require being able to observe processes, resources, files, and connections well enough to transform a production symptom into a technical hypothesis. In real incidents, this capability frequently separates structured troubleshooting from random restart attempts.

5.3.1. Processes and Execution Behavior

Linux provides detailed visibility into processes. This visibility makes it possible to answer questions that directly connect application architecture to physical resource consumption.

An operational investigation typically needs to determine several points:

- **Which process is consuming CPU:** This helps identify loops, expensive parsing, or hot spots;
- **Which process is consuming memory:** The behavior may indicate a leak, excessive cache, or state duplication;
- **How many processes exist:** Unexpected growth may indicate incorrect forking or missing cleanup;
- **What the process tree looks like:** Parent-child relationships help explain servers, workers, and subprocesses;
- **Which user executes each process:** Execution identity matters for security and the principle of least privilege.

An important pattern in AI Engineering appears in servers with multiple workers. If each worker independently initializes a large model, memory footprint can grow approximately with the number of processes. An attempt to increase throughput by increasing worker count can therefore turn into an OOM condition.

This example demonstrates why metrics should not be analyzed in isolation. Diagnosis depends on the relationship among the process model, memory architecture, and serving topology.

5.3.2. Networking, Filesystem, and I/O

A large portion of modern AI systems is distributed. An apparently simple application may simultaneously depend on model APIs, databases, vector databases, object storage, and external tools. For this reason, not every failure perceived by the application originates in the application itself.

Connectivity problems can take different forms:

- **Connection refused indicates an absent or rejecting endpoint:** The application may be trying to access a service that is not listening on that interface or port;
- **Incorrect DNS prevents proper resolution:** A failure occurring before the request can appear to be service unavailability;
- **Timeouts can originate at different layers:** The network, provider, or application may be responsible;
- **Ports may be unavailable:** Configuration or network policy can prevent communication;
- **Connection pools can become bottlenecks:** Saturation limits parallelism even when the backend still has capacity;
- **Sockets can remain stuck or exhaust resources:** Highly concurrent services must manage connections carefully.

If a model call exhibits high latency, the investigation must separate hypotheses involving the provider, DNS, TLS, network, connection pooling, and the application's own logic.

Application observability does not replace an understanding of networking. The same principle applies to storage.

AI Engineering uses files for datasets, checkpoints, logs, indexes, configurations, caches, and other artifacts. However, different storage types provide different semantics and trade-offs:

- **Local disk provides low latency:** However, sharing is limited and the storage is tied to a specific machine;
- **Network filesystems simplify sharing:** However, they introduce network dependencies and latency;
- **Object storage should not be treated like a local filesystem:** Its access patterns and operational semantics are different;
- **File permissions are part of the security boundary:** Incorrect configurations can expose data;
- **Disk capacity remains finite:** Logs, temporary files, and artifacts without retention policies can consume all available space.

When a filesystem reaches full capacity, behavior may appear disconnected from the underlying problem. The application may stop writing logs, creating temporary files, saving checkpoints, or updating indexes. Disk usage is therefore also an availability metric.

5.3.3. Resource Limits and Multidimensional Capacity Planning

Linux can limit different resources consumed by processes. These limits may involve open files, processes, memory, CPU, and stack size. In large-scale applications, any one of these dimensions can become the system's first constraint.

File descriptors are an important example. Sockets also consume file descriptors. An application with thousands of concurrent connections may reach this limit while CPU and memory still appear healthy.

This leads to a fundamental principle: **capacity planning is multidimensional.**

System capacity cannot be reduced to CPU percentage:

- **Connections may saturate first:** The system stops accepting new clients;
- **File descriptors can be exhausted:** New sockets or files can no longer be opened;
- **Threads can reach operational limits:** Creating additional concurrency becomes impossible or inefficient;
- **Processes are also finite resources:** Uncontrolled forks can degrade the host;
- **Memory may constrain scaling before CPU:** This pattern is common when each worker holds large models or caches;
- **Network bandwidth can limit throughput:** Available compute does not help when data cannot arrive quickly enough;
- **Disk and storage throughput can become bottlenecks:** Applications begin spending time waiting for I/O.

Staff-level engineering consists of discovering which resource saturates first under the real workload rather than applying generic scaling heuristics.

5.3.4. Mechanism-Oriented Debugging

Production debugging should begin by identifying the anomalous resource or component rather than immediately modifying code. An initial investigation can follow a relatively simple sequence:

- **Verify that the process is running:** Before investigating performance, confirm that the application is actually executing;
- **Observe CPU pressure:** High utilization may indicate increased load, a regression, a loop, or an expensive algorithm;
- **Observe memory over time:** The temporal pattern helps distinguish legitimate utilization from a leak;
- **Check the filesystem and disk capacity:** Write failures can produce unexpected symptoms;
- **Test dependencies and connectivity:** The application may be healthy while another service is unavailable;
- **Correlate logs with metrics:** Application events help explain changes observed in resource behavior;
- **Compare multiple instances:** A localized failure has different causes from systemic degradation.

This process reduces Mean Time to Recovery because it progressively narrows the hypothesis space. At Staff level, troubleshooting should not depend on restarting and observing what happens.

A restart may temporarily restore service, but it can also destroy a state that would have been useful for understanding the root cause. A memory leak, for example, temporarily disappears after a restart and returns hours later.

The objective is not merely to restore availability. It is to understand the failure mechanism well enough to prevent recurrence.

5.4. Containers, Isolation, and Security

Containers standardize how applications and their dependencies are packaged and executed. They increase consistency across environments, simplify deployment, and integrate naturally with CI/CD pipelines.

However, a container is not a smaller virtual machine. Containers are fundamentally processes isolated through mechanisms provided by the operating system, and they continue to share the host kernel. Understanding this distinction is fundamental to performance, capacity planning, troubleshooting, and cybersecurity.

5.4.1. Docker Images and Layers

A Docker image represents an immutable artifact containing the filesystem, runtime, and dependencies required to execute an application. In AI Engineering, an image may include the Python runtime, libraries, code, tools, and native dependencies.

This model provides relevant operational benefits:

- **Reproducibility reduces differences across environments:** The same artifact can be used in development, staging, and production;
- **Consistency simplifies deployment:** Dependencies become part of the versioned package;
- **Immutability simplifies rollback:** Previous releases can be preserved;
- **Images integrate naturally with CI/CD:** Build, scanning, signing, and publishing can be automated;
- **Dependencies remain isolated from the host:** Applications do not need to rely on libraries installed directly on the machine.

AI workloads, however, can produce extremely large images. ML frameworks, native libraries, CUDA, and potentially model weights can significantly increase artifact size.

Excessively large images affect several dimensions:

- **Builds take longer:** Changes require more time to produce a new artifact;
- **Transfers between registries and nodes become slower:** This can increase deployment time;
- **Cold start can increase:** The workload takes longer to become available;
- **Storage consumption increases:** Registries and nodes must store more data;
- **Attack surface grows:** More packages mean more potentially vulnerable components.

The image should therefore contain what is required at runtime rather than simply reproducing the development environment.

Docker images are organized into layers. This architecture enables caching and reuse during builds, but it also introduces important implications:

- **Stable dependencies can occupy reusable layers:** Code changes do not require a complete rebuild;
- **Layer ordering affects build performance:** Frequently changing elements should be structured to avoid unnecessarily invalidating caches;
- **Layers are immutable:** New modifications create new layers;
- **Removing a file later does not guarantee that it disappeared from history:** Data added to an earlier layer may remain present in the artifact.

This produces a fundamental rule: **secrets must never be embedded in an image.** Adding a credential during the build and removing it in a later step does not guarantee that it has been eliminated. Credentials must be supplied at runtime through appropriate mechanisms.

A container image is part of the software supply chain and must be treated according to the same integrity and provenance criteria applied to other production artifacts.

5.4.2. Namespaces and cgroups

Containers depend heavily on two Linux mechanisms: namespaces and cgroups. Namespaces restrict the view a process has of its environment.

Depending on the configuration, they can isolate different resources:

- **Process namespaces alter visibility into processes:** The container does not need to see every process on the host;
- **Network namespaces isolate interfaces and network stacks:** Containers can have their own virtual topology;
- **Filesystem isolation provides a specific view of files:** The process operates over its mounted environment;
- **Hostname isolation separates system identity:** Each environment can expose its own hostname;
- **User namespaces can contribute to identity isolation:** This helps reduce privilege exposure.

Together, these mechanisms create the experience of an independent environment. However, the kernel remains shared. This distinguishes containers from virtual machines and has security implications.

Good practices must reduce the privileges available to the workload:

- **Run as a non-privileged user to limit impact:** A compromised process has fewer capabilities;
- **Reduce capabilities:** Containers should not receive permissions the application does not use;
- **Use read-only filesystems when possible:** This reduces arbitrary runtime modifications;
- **Avoid privileged containers:** They significantly weaken the isolation boundary;
- **Keep the runtime and kernel updated:** Isolation depends on the security of the underlying layers.

cgroups solve another problem: **competition for resources**. Control groups make it possible to account for and limit resources consumed by processes, including CPU, memory, I/O, and process count.

This capability is fundamental in shared environments:

- **CPU limits restrict how much processing a workload can consume:** This protects other services but can also cause throttling;
- **Memory limits create memory boundaries:** A container may enter OOM before the host does;
- **I/O limits can control pressure on storage:** This reduces interference among workloads;
- **Process limits prevent arbitrary growth:** Forks and subprocesses do not need to be unlimited.

An apparently slow application may be running on a host with available CPU and still experience throttling because its cgroup restricts consumption. Similarly, a container may be terminated due to OOM while the host still has several gigabytes of free RAM.

Troubleshooting must therefore distinguish three dimensions:

- **Physical host capacity:** How many resources the machine possesses;
- **Capacity made available to the container:** How much the workload is authorized to consume;
- **Actual consumption:** How much is being used at that moment.

Ignoring any of these layers can produce an incorrect diagnosis.

5.4.3. Container Networking and Trust Boundaries

Containers typically operate with network namespaces and connect through virtual networks. This layer supports communication among workloads, port exposure, isolation, and policy enforcement. In AI platforms, network topology should reflect the system's trust boundaries.

Consider a simple architecture:

- **The front end needs to access the API:** This does not mean it also needs access to internal databases;
- **The API may need to access retrieval:** Access should be restricted to required dependencies;
- **Retrieval may need to query storage:** That permission does not need to be granted to every component;
- **Only specific services should be able to reach sensitive data:** The existence of physical connectivity does not imply authorization.

Allowing unrestricted communication simply because services are running in the same cluster significantly weakens isolation. **Network segmentation limits blast radius.**

If an agent or service is compromised, that should not automatically grant access to internal databases, the control plane, secret services, or workloads belonging to other tenants. The principle of least privilege must also be applied to the network.

5.4.4. Image Security and the Software Supply Chain

Container security begins before deployment. A compromised image can introduce malicious code even when internally developed code is correct. Container images must therefore be treated as part of the software supply chain.

The main risks can emerge at different points in the build and runtime lifecycle:

- **Base images may contain known vulnerabilities:** Operating system dependencies also have CVEs;
- **Unnecessary packages expand the attack surface:** Every additional component introduces new exploitation opportunities;
- **Running as root increases potential impact:** A vulnerability gains stronger privileges;
- **Embedded secrets can be extracted:** Layers make this risk particularly severe;
- **Unverified images can be tampered with:** The deployed artifact may not correspond to the expected one;
- **Dependencies can be malicious:** Supply-chain attacks can occur during installation or build.

Several practices reduce this surface:

- **Use minimal, trusted base images:** Fewer components mean less surface area to protect;
- **Perform vulnerability scanning:** Known vulnerable dependencies must be identified before deployment;
- **Pin critical versions when appropriate:** Reproducibility depends on controlling dependency changes;
- **Sign artifacts:** Provenance and integrity must be verifiable;
- **Run as a non-root user:** The runtime should operate with the least privilege possible;
- **Update images regularly:** Immutability does not mean preserving vulnerabilities indefinitely.

Container security must be part of CI/CD. Applying security only after a workload has entered production increases cost and reduces the ability to prevent incidents.

5.5. Operating AI Workloads in Containers

AI workloads intensify several operating system problems. A conventional application may have a relatively small runtime. An inference service may simultaneously depend on heavyweight frameworks, drivers, accelerators, large model weights, and tens of gigabytes of memory. For this reason, heuristics that are valid for traditional web serving should not automatically be transferred to LLM serving.

5.5.1. Models, GPUs, Startup, and Readiness

Model weights do not necessarily need to be part of the Docker image. Separating the application lifecycle from the model lifecycle can reduce image size and simplify independent updates. However, this decision transfers part of the cost to startup and model distribution mechanisms.

Several factors must be considered in inference workloads:

- **Model weights can remain external to the image:** This reduces the application artifact size but requires a loading strategy;
- **GPU access must be controlled explicitly:** Containers should receive only the accelerators they require;
- **Drivers remain related to the host:** Compatibility among the host, runtime, and container libraries must be planned;
- **Cold start can be significant:** Loading tens of gigabytes may take a meaningful amount of time;
- **Startup peak memory may exceed steady state:** Memory limits must accommodate initialization;
- **Readiness must represent the actual ability to serve traffic:** A started process does not mean the model is loaded.

This final distinction is critical. A container may be alive while loading weights or initializing internal structures. If traffic is routed to it during this period, the system may produce timeouts, errors, or high tail latency.

Liveness and readiness answer different questions. The existence of the process does not prove that the service is ready to perform inference.

5.5.2. Scaling Large Models

Model serving combines processes, containers, and accelerator memory. Creating more containers or workers does not guarantee higher throughput. If every replica must load an independent copy of the model, horizontal scaling also multiplies RAM or VRAM consumption. This can make scaling prohibitively expensive or physically impossible.

Several strategies emerge in this context:

- **Batching increases the amount of work performed per execution:** It can improve accelerator utilization but increases memory pressure and latency;
- **Tensor parallelism distributes parts of the computation:** It enables execution of models requiring the combined capacity of multiple devices;
- **Model parallelism distributes model components:** It may be necessary when a single GPU cannot hold the complete execution;
- **Multiple GPUs expand available resources:** However, they introduce complexity and cost;
- **Independent replicas increase serving capacity:** Each replica may also require its own copy of the weights.

The choice depends on model size, traffic pattern, available memory, and throughput and latency objectives. A conventional HTTP service may scale through many small replicas. A model containing tens of billions of parameters has a different cost and resource structure. The serving architecture must reflect that difference.

5.5.3. Relationship Among Hardware, Linux, Containers, and Kubernetes

Kubernetes does not replace the operating system or containers. It coordinates workloads that continue to execute through these layers.

The conceptual relationship can be understood as a stack:

- **Hardware provides CPU, RAM, disk, network, and GPU:** These are finite physical resources;
- **Linux manages these resources:** Processes, memory, and I/O are controlled by the operating system;
- **The container runtime creates isolated processes:** Namespaces and cgroups implement much of this abstraction;
- **Kubernetes determines placement and lifecycle:** The orchestrator determines where workloads execute and attempts to maintain the desired state;
- **The application operates on top of all these layers:** Observed behavior can be limited by any one of them.

This decomposition is extremely useful during incident response. When a pod exhibits high latency, several causes are possible:

- **The application itself may be slow:** An algorithm or implementation may have regressed;
- **The container may be incorrectly configured:** Startup or dependencies may be failing;
- **A resource limit may cause throttling:** The application requires more CPU than it is authorized to consume;
- **The node may be under pressure:** Other workloads compete for the same resources;
- **The network may be the bottleneck:** Remote dependencies may introduce latency;
- **Storage may be slow:** The process spends time waiting for data;
- **Scheduling may place workloads incorrectly:** Resources actually available may not match workload requirements.

Jumping directly to the code ignores a large part of the possible failure space.

5.5.4. Resource Management, Isolation, and FinOps

Requests and limits should reflect observed workload behavior. Arbitrary values create two symmetric classes of problems: **underprovisioned resources and overprovisioned resources.**

When resources are insufficient, the system loses performance or availability:

- **Insufficient CPU increases latency:** The process may experience contention or throttling;
- **Insufficient memory produces OOM:** Containers may repeatedly restart;
- **Insufficient GPU resources prevent model loading:** The deployment may never become ready;
- **Insufficient I/O limits pipelines:** Ingestion, training, or serving spend more time waiting for data.

Overprovisioning also has a cost:

- **Infrastructure remains idle:** The organization pays for capacity without converting that resource into throughput;
- **Scheduling becomes less efficient:** Workloads reserve larger nodes than they actually require;

- **Cluster density decreases:** Fewer services fit within the same infrastructure;
- **Product margins deteriorate:** Fixed or variable cost per workload increases.

Capacity planning should therefore use historical metrics and actual workload behavior. This process connects Platform Engineering directly to FinOps.

Resource isolation is also not merely a security mechanism. It is a reliability mechanism.

Consider a cluster simultaneously running a critical API, a batch pipeline, and an experimental service. Without appropriate boundaries, the experimental workload may consume enough CPU and memory to degrade the critical system.

cgroups and containers help limit this behavior:

- **Predictability increases:** Each workload has a clearer resource budget;
- **Reliability improves:** An anomalous process has less ability to affect others;
- **Capacity planning becomes more precise:** Consumption can be measured and attributed;
- **Multi-tenancy becomes operationally safer:** One tenant should not be able to degrade all others.

This pattern is especially important for preventing the **noisy neighbor problem**.

5.5.5. Debugging CPU, Memory, and I/O

When a service exhibits high CPU utilization, the response should not automatically be to increase capacity. The responsible mechanism must first be identified.Possible causes include loops, expensive parsing, serialization, compression, excessive thread counts, busy waiting, or inefficient algorithms.

Several questions help distinguish among hypotheses:

- **Has traffic increased:** In that case, higher consumption may be proportional to demand;
- **Has CPU per request increased:** This suggests a change in unit cost and a possible regression;
- **Does a single process dominate CPU:** A concentrated hotspot may exist;
- **Are many processes competing simultaneously:** Excessive parallelism may itself be the problem;
- **Is the container being throttled:** The application may require capacity that the cgroup does not allow it to use.

Scaling may be appropriate when real demand has increased. However, scaling inefficient code merely multiplies waste.

Memory debugging requires similar reasoning. High utilization does not automatically mean a memory leak. An inference service may legitimately keep a model or cache in memory.

The temporal pattern is more informative:

- **Memory grows and stabilizes:** This may represent expected behavior;
- **Memory grows continuously:** This is consistent with a leak;
- **Memory grows with traffic and does not return:** State may be retained improperly;
- **Each worker adds a large amount of memory:** Model or data structure duplication may be occurring;
- **Spikes coincide with batches:** Batch size may exceed planned capacity.

Caches require particularly careful control. Every long-running cache needs at least policies related to:

- **Limit:** How large it is allowed to grow;
- **Expiration:** How long a given entry remains valid;
- **Eviction:** Which content is removed when the limit is reached.

An optimization designed to reduce latency must not compromise availability. I/O completes the diagnosis.

A service can exhibit low CPU utilization while still having high latency. In that case, the application may be waiting for disk, object storage, a database, or the network.

In document pipelines, processing may stall while waiting for file transfers. During training or inference, GPUs may remain idle while waiting for data. This directly connects performance to economics: **idle expensive hardware is also waste.**

A GPU that remains underutilized because the data pipeline cannot feed it is an architectural and economic problem, not merely an infrastructure metric.

5.5.6. Cybersecurity, Sandboxing, and Untrusted Code

The security of AI systems depends heavily on the security of the environment in which they execute. Several configurations unnecessarily increase the blast radius of a compromise:

- **Processes running with excessive privileges increase attacker capabilities:** The compromised service gains permissions it did not require to function;
- **Containers running as root increase impact:** An exploit operates with stronger privileges inside the environment;
- **Unnecessary host filesystem mounts expose sensitive resources:** A container may access data that should remain outside its boundary;
- **An exposed Docker socket grants extremely powerful capabilities:** The workload may obtain control approaching that of the host;
- **Unrestricted outbound networking facilitates exfiltration:** A compromised process can communicate freely with external destinations;
- **Secrets without adequate controls can leak:** Environment variables and files also require appropriate policies;
- **Outdated base images retain known vulnerabilities:** Immutability does not replace patching;
- **Untrusted code execution can compromise the entire environment:** This is especially important for coding agents and computer-use agents.

If an agent can execute arbitrary code, the infrastructure must assume that the code may be malicious. The sandbox must restrict several resources simultaneously:

- **Resources must be limited:** Untrusted code should not be able to consume CPU or memory indefinitely;
- **The filesystem must be restricted:** Only required files should be available;
- **Network egress must be controlled:** Code should not be able to arbitrarily reach internal networks or the Internet;
- **Credentials must be minimal:** The environment should not contain unnecessary secrets;
- **Lifetime must be limited:** Sandboxes do not need to remain available indefinitely.

In this context, sandboxing is not merely a stability mechanism. It is a security boundary. Containers provide meaningful isolation but do not constitute the strongest available boundary because they share the host kernel.

Depending on the threat model, highly untrusted workloads may require additional isolation:

- **Virtual machines provide stronger isolation:** Operational and startup costs tend to be higher;
- **MicroVMs can provide an additional boundary:** They also introduce complexity and overhead;
- **Sandbox runtimes can reduce host exposure:** The choice depends on risk and workload characteristics;
- **Dedicated hosts increase separation:** In exchange, they reduce resource-sharing efficiency.

The trade-off is deliberate:

- **Stronger isolation reduces blast radius:** Escaping one layer has less potential for propagation;
- **Stronger isolation adds overhead:** Startup, memory, and cost can increase;
- **Containers provide excellent efficiency for many workloads:** Especially when the code is trusted;
- **User-controlled code requires stricter criteria:** The lower the trust level, the stronger the boundary should be.

Architecture must be determined by the threat model, not by tool convenience.

5.6. Staff-Level Mindset for Operating Systems and Containers

Staff-level operating system knowledge is not measured by how many Linux commands someone has memorized. Its value lies in using these mechanisms to make better architectural decisions, diagnose failures faster, and connect physical resource utilization to reliability, security, and economics.

5.6.1. Technical Trade-offs

There is no universally superior architecture for executing AI workloads. Every decision favors certain properties while weakening others.

The central trade-offs in this chapter can be summarized as follows:

- **Processes vs. Threads:** Processes increase isolation and memory independence, while threads share state with lower overhead;
- **Parallelism vs. Coordination Cost:** More workers increase throughput only while the additional useful work exceeds scheduling, context switching, and communication overhead;
- **Memory Usage vs. Performance:** Caches and preloading can reduce latency but increase consumption and OOM risk;
- **Isolation vs. Efficiency:** Containers use resources efficiently, while VMs provide stronger boundaries;
- **Resource Limits vs. Burst Capacity:** Limits protect workloads from one another but may prevent legitimate use of idle capacity;
- **Small Images vs. Developer Convenience:** Minimal images reduce transfer cost and attack surface but may provide fewer debugging tools;
- **Local Disk vs. Shared Storage:** Local disk generally provides better latency, while shared storage improves data mobility and availability;
- **More Replicas vs. Model Memory:** Horizontal scaling increases capacity, but each replica may multiply large-model memory consumption;
- **Batch Size vs. Latency and Memory:** Larger batches can improve compute utilization while increasing waiting time and peak memory;
- **Stronger Sandboxing vs. Operational Overhead:** Additional isolation reduces cyber risk but adds cost and complexity.

The objective is not to eliminate trade-offs. It is to make them explicit and consciously choose which properties must be prioritized for each workload.

5.6.2. Operational Anti-Patterns

Immature environments frequently exhibit patterns that appear locally convenient but accumulate systemic risk. Several should be identified quickly during design reviews and incident response:

- **Running every service as root:** Unnecessarily increases the impact of any compromise;
- **Running containers without limits:** A single workload can consume capacity required by others;
- **Building unnecessarily massive images:** Cold start, build time, storage consumption, and attack surface all increase simultaneously;
- **Storing secrets inside an image:** Credentials may remain in layers even after being removed;
- **Adding workers indefinitely:** Excessive parallelism can reduce performance;
- **Creating unbounded caches:** OOM eventually becomes a consequence of the design;
- **Duplicating models across multiple processes without planning:** RAM or VRAM can be exhausted quickly;
- **Treating disk as infinite log storage:** The filesystem eventually reaches full capacity;
- **Treating a running process as sufficient readiness:** A workload may be alive while still unable to serve requests;
- **Treating containers as absolute sandboxes for hostile code:** The isolation level may be inadequate for the threat model;
- **Restarting services as the only troubleshooting technique:** The symptom disappears while the root cause remains.

These patterns share a common characteristic: **they solve or hide a local problem by transferring cost or risk to another part of the architecture.** Recognizing them early creates engineering leverage.

5.6.3. Business Impact

Infrastructure may appear distant from the product, but operating system decisions directly affect economics and user experience. An efficient architecture generates both technical and financial benefits:

- **Higher CPU utilization reduces required infrastructure:** More work can be executed on the same hardware;
- **Lower memory footprint increases density:** More workloads can share the same capacity;
- **Standardized containers reduce time-to-market:** Deployments become more reproducible and predictable;
- **Resource isolation reduces shared incidents:** An anomalous service has less ability to bring down the entire platform;
- **Smaller images accelerate releases:** Faster builds and distribution improve developer velocity;
- **Capacity planning reduces waste:** Resources can be provisioned closer to actual demand;
- **Appropriate sandboxes reduce cyber risk:** This is particularly important for agentic systems capable of executing actions;
- **More efficient troubleshooting reduces MTTR:** Less downtime means lower operational and financial impact;
- **Better accelerator feeding increases economic utilization:** GPUs spend less time idle while waiting for data;
- **Lower consumption per request improves unit economics:** Operational efficiency can directly alter margins.

In computationally expensive AI products, small percentage improvements in utilization can produce large economic differences at scale. This brings Operating Systems directly into the domain of FinOps.

The Staff AI Engineer must therefore learn to translate CPU, RAM, VRAM, throughput, cold start, and resource utilization into a broader question: **how much does it cost to produce one unit of business value reliably?**

Chapter Summary

Operating systems and containers form the layer where AI applications actually compete for physical resources. Models and APIs may dominate architectural discussions, but reliability and economics are frequently determined by processes, memory, I/O, resource limits, and isolation.

The main principles of this chapter can be summarized as follows:

- **Choose processes and threads according to the workload and required isolation:** There is no universal concurrency strategy;
- **Do not increase parallelism without measuring coordination and physical capacity:** More workers do not automatically mean higher throughput;
- **Treat data movement as part of the cost:** Parallel compute can lose value when large volumes of data must be copied between processes;
- **Treat memory as a critical resource:** OOM conditions can produce more abrupt failures than high CPU utilization;
- **Capacity planning must account for peak memory:** Averages hide the spikes responsible for incidents;
- **Understand Linux well enough to diagnose production systems:** Processes, networking, filesystems, and limits are part of the AI Engineering toolkit;
- **Capacity planning is multidimensional:** CPU, memory, sockets, file descriptors, I/O, and network can all constrain the system;
- **Containers are isolated processes, not complete VMs:** A shared kernel has security implications;
- **Use namespaces and cgroups consciously:** They underpin isolation and resource control;
- **Design container images as production artifacts:** Size, vulnerabilities, secrets, and provenance matter;
- **Separate the model lifecycle from the application lifecycle when that improves operations:** Model weights do not necessarily need to be embedded in the image;
- **Do not automatically transfer web-serving heuristics to LLM serving:** Model memory and accelerator economics change the scaling strategy;
- **Readiness must represent actual capability:** A live process does not mean a service is ready to receive traffic;
- **Use sandboxing proportional to risk:** Untrusted code may require stronger boundaries than conventional containers;
- **Connect resource utilization to cost:** Operational efficiency directly affects unit economics.

The most important lesson is recognizing that many problems initially classified as "application problems" actually emerge from interactions among code, processes, memory, networking, storage, and execution environment limits. The greater the seniority, the less acceptable it becomes to treat these layers as a black box.

Practical Framework for Diagnosing Execution Problems

When investigating an AI system experiencing performance, availability, or stability problems, the Staff AI Engineer should begin by classifying the symptom before modifying the architecture. A diagnostic sequence narrows the hypothesis space and prevents unnecessary scaling or code changes. First, analyze CPU:

- **Is utilization high or low:** High CPU and low CPU suggest different categories of bottlenecks;
- **Is the workload actually CPU-bound:** High latency does not necessarily mean insufficient CPU;

- **Are there too many processes or threads:** Excessive concurrency can degrade throughput;
- **Is the container being throttled:** Physical host capacity does not mean capacity is available to the workload;
- **Does the increase correlate with traffic:** Proportional growth may indicate legitimate demand rather than regression;
- **Has CPU per request changed:** Increased unit cost suggests a change in application behavior.

Next, analyze memory:

- **How much does each process consume:** This helps identify duplicated models or caches;
- **Is there continuous growth:** Monotonic behavior may indicate a leak;
- **Do caches have limits and eviction policies:** Uncontrolled caches turn growth into OOM;
- **Are there multiple copies of the same model:** Worker count can multiply memory footprint;
- **Can the container accommodate peak memory:** Startup or batches may require more memory than steady state;
- **Is batch size pressuring RAM or VRAM:** Additional throughput may be purchased at the cost of availability risk.

Then, analyze I/O and dependencies:

- **Is the process waiting for network, disk, storage, or a database:** Low CPU combined with high latency often indicates waiting at some layer;
- **Are there timeouts or saturated connections:** Connection pooling and network limits can constrain throughput;
- **Is storage responding within expectations:** Data pipelines depend on how quickly data arrives;
- **Is disk usage close to the limit:** A full filesystem can interrupt logs, temporary files, and checkpoints;
- **Are GPUs waiting for data:** Accelerator utilization must be analyzed together with the pipeline feeding the hardware.

Next, evaluate the containerization layer:

- **Do CPU and memory limits match the workload:** Incorrect configuration can produce throttling or OOM;
- **Does the workload use least privilege:** Security must be part of the runtime;
- **Does the image contain only required components:** Size and attack surface must be controlled;
- **Are there known vulnerabilities:** Supply-chain risk cannot be ignored;
- **Do readiness and liveness represent the correct states:** A started process and an available service are not equivalent;
- **Does the network configuration respect trust boundaries:** Connectivity should follow least privilege.

For AI workloads, add workload-specific questions:

- **Does the model comfortably fit within available memory:** Operating close to the limit increases fragility;
- **Does every replica need to load a complete copy:** Horizontal scaling may multiply RAM or VRAM consumption;
- **Is batch size optimizing or pressuring resources:** Throughput must be evaluated together with memory and latency;
- **Is the GPU being utilized or waiting for the data pipeline:** Idle hardware represents cost;
- **Does cold start affect availability:** Loading time must be reflected in the deployment strategy;
- **Do model weights need to be inside the image:** Separating artifacts may improve lifecycle management;
- **Is code executed by the agent trusted:** Otherwise, sandboxing must be treated as a security boundary.

Finally, connect the diagnosis to the business:

- **Does the problem increase user-perceived latency:** Technical performance must be connected to user experience;
- **Is throughput being reduced:** Lower capacity may constrain revenue or productivity;
- **Is the system causing restarts or incidents:** Reliability has operational cost;
- **Is there significant infrastructure waste:** Idle or overprovisioned resources degrade economics;
- **Does the change reduce cost per request or task:** Optimizations must be evaluated economically;
- **Is there a security risk in the current environment:** The architecture must account for the potential impact of compromise.

This is the mental model expected of a Staff AI Engineer: **understand the execution infrastructure deeply enough to transform operational symptoms into technical mechanisms and, from those mechanisms, make decisions that improve system reliability, performance, security, and economics.**