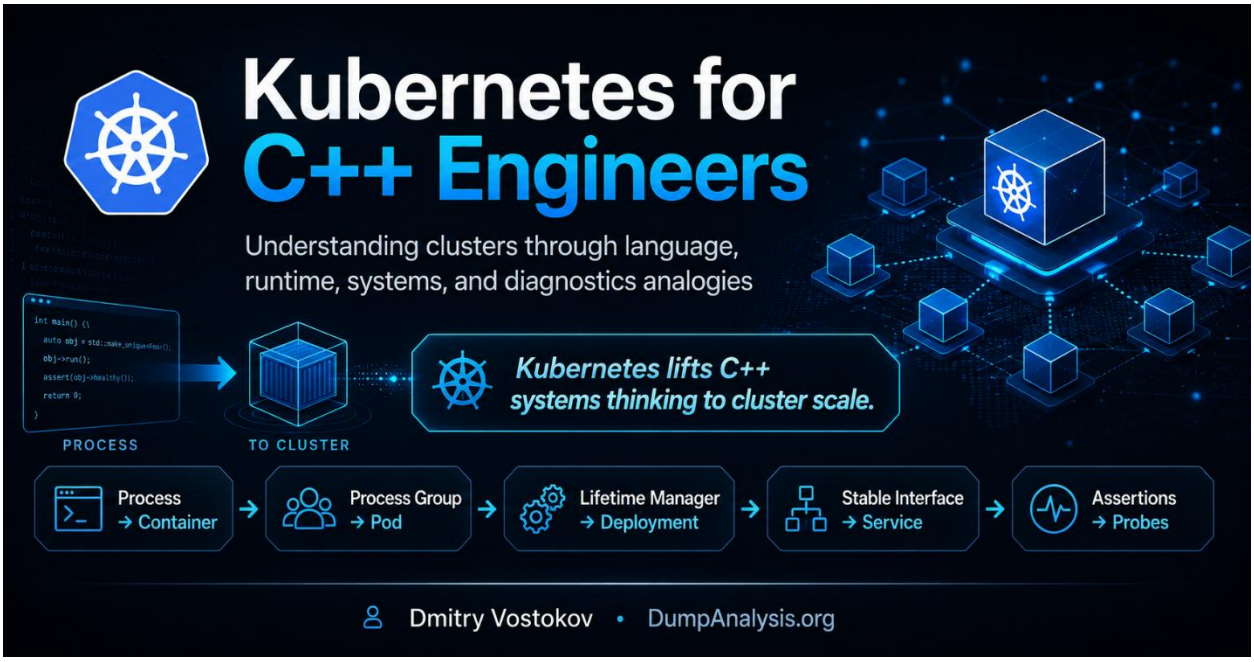
Kubernetes for C++ Engineers

Understanding clusters through language, runtime, systems, and diagnostics analogies

Dmitry Vostokov (DumpAnalysis.org)
with GPT-5.5 and Claude Sonnet 4.6 (AI-assisted drafting)



Contents

Introduction: not YAML, but distributed systems thinking

A C++ engineer approaching Kubernetes often has two simultaneous reactions. The first is recognition: containers look like processes, resource limits look like memory constraints, Services look like stable interfaces, Namespaces look familiar, and the whole thing feels like a large runtime. The second is discomfort: Kubernetes is declarative, eventually consistent, network-heavy, YAML-driven, and full of controllers that change the system behind your back.

The most useful way to understand Kubernetes is not to treat it as “deployment tooling” or as “YAML for Docker.” Kubernetes is easier to understand as a distributed runtime for processes. The C++ engineer already knows the right questions: What is the executable? What owns its lifetime? Where does it run? What resources may it consume? What interface exposes it? What invariant keeps it valid? What happens when it crashes? How do we reconstruct what happened after the fact? Kubernetes answers the same questions at cluster scale.

The central claim of this article is therefore simple: Kubernetes lifts familiar C++ systems concerns — process lifetime, ownership, allocation, invariants, interfaces, compatibility, and diagnostics — from a single process or host into a distributed orchestration space. It does not abolish systems thinking. It changes its scale.

In C++, you write source code, compile it, link it, run it, allocate memory, handle failure, and manage object lifetime. In Kubernetes, you write manifests, build images, deploy containers, expose them through Services, assign resources, recover from failures, and manage application lifetime across a cluster. The conceptual bridge is strong, but not perfect. Kubernetes is not “C++ for servers.” It is a control system for distributed execution. Still, many Kubernetes ideas become much easier to understand when mapped to concepts C++ engineers already know deeply.

The running example: a C++ payment service

To avoid turning Kubernetes into a catalog of object kinds, we will follow one concrete application through the article: a C++ HTTP service called payment-service. The service reads configuration, connects to a database, listens on port 8080, exposes readiness and liveness endpoints, writes logs, and must be safely updated from version 1.0 to 1.1.

```
int main() {
    Config config = load_config();
    Server server(config.port);

    server.initialize_database(config.db_connection);
    server.start();

    return server.run_event_loop();
}
```

In a traditional deployment, we might run this as a binary on a host:

```
./payment-service --config payment.conf
```

In Kubernetes, the binary is still there. Kubernetes does not replace the process. It surrounds the process with packaging, placement, naming, health checking, configuration injection, restart behavior, rollout strategy, security policy, and diagnostic surfaces. The transformation looks like this:

```
C++ source
  -> compiled binary
  -> container image
  -> Deployment creates Pods
  -> Pods run containers
  -> containers run the C++ process
  -> Service gives a stable address
  -> Ingress or Gateway exposes public routes
  -> controllers maintain the desired state
```

This running example is deliberately ordinary. That is the point. Kubernetes is not only for exotic distributed systems. Even a single native service becomes part of a distributed runtime once Kubernetes assumes ownership of its lifecycle.

Part I — From C++ Binary to Kubernetes Workload

Native-code concerns do not disappear inside containers

For C++ engineers, the first discipline is to remember that containers do not make native-code realities disappear. They package those realities. A container image may appear to be a deployment artifact. However, it still contains a binary, shared libraries, dynamic loader assumptions, certificate stores, time zone files, locale files, startup commands, and filesystem layout. The process inside the container may still crash because of a missing shared library, an incompatible libstdc++ version, an unsupported

CPU instruction, or a bad environment variable. It may also contain a stripped binary with no useful symbols.

This is where C++ experience becomes unusually valuable. A Kubernetes incident may eventually reduce to familiar native-code questions: Did the process receive SIGTERM? Did it handle PID 1 behavior correctly?[i] Did it flush logs before termination? Was a core dump produced? Was the image too minimal to contain diagnostic tools? Was the binary built for the correct architecture? Did a custom allocator or memory cache interact badly with the container memory limit?

Distroless and minimal images are excellent for production hygiene, but they make ad hoc debugging harder. Stripped binaries reduce image size, but they make postmortem analysis weaker unless symbols are stored elsewhere. A container memory limit may turn a C++ cache or memory pool from an optimization into an OOM-kill trigger. Kubernetes moves the process into a controlled environment; it does not repeal the rules of native execution.

From executable to container image

A C++ program usually begins as source code. The compiler turns translation units into object files. The linker combines them into an executable. That executable is then loaded by the operating system and becomes a process. In Kubernetes, the usual deployment artifact is a container image.

A container image is not exactly an executable. It is more like an executable plus its runtime filesystem, dynamic libraries, environment assumptions, startup command, and packaging metadata. For a C++ engineer, a container image is closer to a statically described runtime environment than to a single binary.

C++ World	Kubernetes World
Source files	Application source repository
Compiler	Build system/CI pipeline
Object files	Intermediate build artifacts
Linked executable	Application binary inside image
Runtime libraries	Image layers and base image
Executable package	Container image
Process	Running container
Process group	Pod

A container image answers the question: what exact filesystem and startup command should be used to create this application process? That makes it like an executable plus a controlled miniature user space. A poorly built C++ executable can fail at runtime due

to a missing shared library. A badly built container image can fail for the same reason, except now the failure happens inside a container scheduled somewhere in the cluster.

Container as process, Pod as local execution capsule

The most immediate analogy is that a container is like a process. This is useful, but incomplete. A container is a process with isolation boundaries: namespaces, cgroups, filesystem layers, environment variables, and security context. From the application's perspective, it may appear to be running on its own small machine. From the host's perspective, it is still one or more processes.

In Kubernetes, however, the smallest deployable compute unit is not the individual container. It is the Pod. A Pod is a group of one or more containers that are scheduled together onto the same node and share certain resources, especially network identity and optionally volumes. For a C++ engineer, a Pod is like a tightly coupled process group, a local execution capsule, or an application-specific logical host.

```
Pod
  |-- container: payment-service
  |-- container: log-sidecar
  `-- container: metrics-sidecar
```

Containers inside the same Pod are colocated. They run on the same Kubernetes node and share the Pod's network namespace[ii]. One container can talk to another using localhost. This is similar to tightly coupled processes on the same machine, not independent services spread across a network. A Pod is, therefore, like a small addressable execution unit containing cooperating runtime components.

Containers as process-like and thread-like: two diagnostic levels

The usual technical analogy says that a container is close to a process, and a Pod is close to a process group. This is the safest operating-system-level interpretation. However, there is another useful analogy for C++ engineers: from a Kubernetes orchestration perspective, we can sometimes treat the Pod as a process-like entity and the containers within it as thread-like cooperating components.

This thread analogy is not about implementation. A sidecar container is not a C++ thread. It has its own process tree, filesystem view, image, entry point, memory space, and failure modes. It cannot directly access the heap of the main application container. It does not share C++ globals, pointers, stack frames, locks, or ordinary in-process synchronization primitives. The analogy only says that, from above, several containers in a Pod may behave like cooperating components of a single local runtime unit.

This is particularly helpful for sidecars. A sidecar container may collect logs, proxy traffic, refresh configuration, export metrics, or perform auxiliary local work. It supports

the main application, belongs to the same local service, and is not scheduled independently as a separate top-level workload. It shares Pod placement, network identity, and Pod-level ownership, although individual containers may still have distinct startup, restart, probing, and termination behavior.

The correct model is therefore two-level rather than single-level:

View	Container	Pod
Operating-system diagnostic view	Process-like execution unit	Process group/logical host/ execution capsule
Kubernetes orchestration view	Thread-like cooperating component	Process-like schedulable runtime unit

The first view is appropriate when debugging Linux execution, process exits, signals, dynamic libraries, memory limits, core dumps, file descriptors, and container runtime behavior. The second view is appropriate when reasoning about Kubernetes scheduling, ownership, colocation, readiness, restart behavior, and the unit that higher-level controllers manage.

The practical design rule is simple: put multiple containers in a single Pod only when they are as cohesive as helper processes or cooperating threads within one local service. They should share placement, local communication context, and operational ownership. If they need to scale, fail, deploy, or be owned independently, they should probably be separate Pods connected via Services or another explicit communication mechanism.

Part II — Lifetime, Placement, and Reconciliation

Deployment as object lifetime manager

C++ engineers think constantly about lifetime. Who owns this object? Who deletes it? Is this pointer borrowed? Is this resource released? Is this object stack-allocated, heap-allocated, reference-counted, or owned by a container? Kubernetes has a similar, but distributed, concept of lifetime.

A Pod can be created directly[iii], but production systems rarely manage individual Pods manually. Instead, a higher-level object, such as a Deployment, owns the desired workload shape. A Deployment says: “I want N replicas of this application version running.” Kubernetes then continuously tries to make that true.

```
std::vector<std::unique_ptr<Worker>> workers;

for (int i = 0; i < 3; ++i) {
    workers.push_back(std::make_unique<Worker>());
}
```

In Kubernetes, the equivalent lifetime declaration is a Deployment that owns the desired replica set:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: payment-service
  template:
    metadata:
      labels:
        app: payment-service
    spec:
      containers:
        - name: payment-service
          image: example/payment-service:1.0
```

The Deployment does not merely start three Pods once. It keeps trying to maintain three running Pods. If one crashes, Kubernetes creates another. If a node dies, Kubernetes reschedules Pods to other nodes. If you change the image version, Kubernetes performs a rollout. This is RAII-like, but with a major difference: C++ RAII is usually deterministic, whereas Kubernetes object lifetimes are declarative and eventually consistent. A Deployment is not a collection of std::unique_ptr. It is more like a distributed owner with a background repair loop.

Desired state and reconciliation as invariant maintenance

C++ engineers are familiar with invariants. A class has an invariant that should remain true before and after public method calls. A data structure maintains ordering, ownership, or consistency rules. A lock protects a critical section. An assertion checks that the program has not entered an impossible state.

Kubernetes is built around desired-state invariants. You do not usually say: start Pod A, then start Pod B, then restart Pod A if it fails, then move Pod B if its node dies. Instead, you say: there should be three replicas of this application, using this image, with these

labels, exposing this port, and constrained by these resource requirements. The control plane then runs controllers that compare the desired state with the actual state and act on the difference.

```
while (true) {
    DesiredState desired = read_desired_state();
    ActualState actual = observe_cluster_state();

    Diff diff = compare(desired, actual);
    apply_changes(diff);

    sleep_or_wait_for_event();
}
```

For a C++ engineer, this is one of the most important conceptual shifts. Kubernetes is not primarily an API for executing commands. It is an API for declaring a durable desired state. Controllers are the runtime machinery that attempts to make that state true. A successful manifest apply means the desired state was accepted. It does not mean the actual workload is already running, reachable, ready, or healthy.

The control plane is a declarative runtime and supervisor

The Kubernetes control plane has several major components: the API server, scheduler, controllers, and etcd. For a C++ engineer, these components are best understood as a declarative runtime and supervisory system, rather than a direct equivalent of a compiler or an operating system kernel.

Some parts resemble compilation: manifests are validated, admitted, defaulted, and stored. Some parts resemble allocation: the scheduler places Pods onto nodes. Some parts resemble runtime supervision: controllers and kubelets observe actual state and act on drift. But the control plane is not a compiler in the C++ sense. It does not produce a binary. It maintains a distributed object graph.

Kubernetes Component	Systems Analogy
API server	Runtime API and validation/admission boundary
etcd	Authoritative persistent state store
Scheduler	Placement algorithm/cluster allocator
Controller manager	Runtime invariant maintenance
Kubelet	Per-node local supervisor
Container runtime	Low-level process launcher and container backend

The scheduler is particularly familiar to systems programmers. It is not an OS thread scheduler; it does not decide which CPU instruction runs next. It decides where a Pod should run, based on resource requests, constraints, affinity rules, taints, tolerations, and current cluster state. It is a placement allocator for workload execution sites.

Nodes, scheduling, and resource allocation

A Kubernetes Node is a worker machine: a physical server, virtual machine, or cloud instance. It runs Pods. It has CPU, memory, storage, networking, kubelet, and a container runtime. A Node is not like a C++ object instance. It is closer to a host OS environment capable of running many processes.

A cluster is therefore not one process and not one machine. It is a coordinated collection of machines running many application processes under control-plane supervision. This matters because many C++ intuitions are local. You may be used to memory, threads, locks, file descriptors, and crashes within a single process or host. Kubernetes forces you to think about partial failure. A Pod may be running, terminating, restarting, unreachable, pending, or rescheduled. A Service may exist even when no backing Pods are ready[iv]. A node may be alive from one perspective and unreachable from another.

Resource requests and limits make this placement explicit. A request says: this container needs at least this much CPU or memory for scheduling purposes[v]. A limit says: this container must not exceed this amount. For memory, exceeding the limit results in termination rather than a graceful C++ allocation failure. For CPU, limits may produce throttling. Kubernetes sees container-level consumption; it does not understand your allocator, memory pool, cache, or background worker intent.

```
resources:
  requests:
    cpu: "500m"
    memory: "256Mi"
  limits:
    cpu: "1"
    memory: "512Mi"
```

Rollouts, StatefulSets, DaemonSets, Jobs, and CronJobs

A Deployment rollout is a controlled replacement of runtime instances. Updating the payment service from version 1.0 to version 1.1 gradually replaces old Pods with new Pods while preserving the stable Service interface. This resembles replacing implementation objects behind an interface, but in a distributed environment with mixed versions.

C++ Concern	Kubernetes Rollout Concern
ABI compatibility	Protocol and schema compatibility
Function signature change	API contract change
Object layout change	Data format change
Dynamic library mismatch	Mixed-version service interaction
Undefined behavior	Inconsistent distributed behavior

A safe rollout requires the same discipline as safe binary compatibility: versioning, backward compatibility, staged migration, observability, and rollback strategy. Kubernetes can move traffic and replace Pods, but it cannot infer that version 1.1 changed a database schema too early or started speaking a subtly incompatible protocol.

A StatefulSet is different. It is used when identity matters, such as in databases, consensus systems, queues, clustered storage, or applications where each replica has a distinct role. A Deployment is like a vector of fungible workers. A StatefulSet is more like an indexed array where node[0], node[1], and node[2] have stable identities and storage associations[vi].

A DaemonSet ensures that a copy of a Pod runs on each selected node[vii]. This is like installing a local helper daemon on every machine: a log collector, a metrics agent, a network plugin, a storage component, or a security monitor. A Job runs a task to completion, like a command-line utility whose main() is expected to return successfully. A CronJob runs such work on a schedule. The distinction matters diagnostically: if a Deployment finishes, that is usually a failure; if a Job exits successfully, that is the intended result.

Part III — Interfaces, Boundaries, and Runtime Configuration

Labels and selectors as runtime traits and queries

Labels are key-value pairs attached to Kubernetes objects. Selectors are queries over labels. A Service, Deployment, NetworkPolicy, or other object can select Pods based on labels. For C++ engineers, labels resemble type traits, tags, annotations, or concepts, except they exist at runtime in the cluster object model rather than at compile time.

```
metadata:
  labels:
    app: payment-service
    tier: backend
    version: v1
```

Labels are not inheritance and not strict typing. They are metadata-based classification. That flexibility is powerful and dangerous. A wrong label can silently disconnect traffic from Pods or route traffic to the wrong Pods. For C++ engineers, labels are like a dynamic type classification system for distributed runtime objects. They require naming discipline.

Service as a stable interface

Pods are ephemeral. Their names, IP addresses, and lifetimes are unstable. If a Pod crashes and is replaced, the replacement Pod has a different identity. Other components need a stable way to reach a changing set of Pods. That is the purpose of a Kubernetes Service.

A Service provides a stable network identity and load-balancing abstraction across a set of Pods selected by label selectors. For a C++ engineer, a Service is like a stable interface that hides changing implementation details.

C++ Concept	Kubernetes Concept
Interface	Service
Implementation objects	Pods
Dynamic dispatch	Service load balancing
Object replacement	Pod rescheduling
Stable symbol name	DNS name of Service

However, a Service is not an interface in the type-system sense. It does not check method signatures. It does not know your protocol semantics. If payment-service points to Pods that do not actually speak the expected protocol, Kubernetes will not save you. It is more like linking successfully to a function with the wrong semantic contract.

Ingress and Gateway as public API boundaries

A Service usually provides access inside the cluster. To expose applications to the outside world, Kubernetes commonly uses Ingress or Gateway-style abstractions, depending on the environment. For C++ engineers, this resembles a public API boundary. Inside a library, you may have many internal classes and private headers, but you expose a carefully controlled public interface. Inside a Kubernetes cluster, you may have many internal Services, but only some routes should be exposed externally.

```
https://example.com/payments
        -> Ingress or Gateway route
        -> payment-service Service
        -> payment-service Pods
```

The analogy highlights an important discipline: do not expose everything. Good Kubernetes architecture, like good C++ library design, separates public contracts from internal implementation details.

Gateway API is commonly used in modern clusters, but it is an add-on API family implemented by gateway controllers rather than a primitive workload object like Pod or Deployment.

ConfigMap, Secret, and environment-specific runtime state

C++ programs often distinguish between compiled constants, runtime configuration, environment variables, and sensitive credentials. Kubernetes has ConfigMaps and Secrets. A ConfigMap stores non-sensitive configuration data. A Secret stores sensitive data such as passwords, tokens, and certificates, but engineers should not treat the name “Secret” as magic. Secret security depends on cluster configuration, access control, encryption at rest, careful mounting, and operational discipline.

A ConfigMap is like a file of externalized runtime constants. A Secret is similar in shape but different in intent: runtime configuration with confidentiality expectations. The design rule is that the image should usually be generic, while environment-specific values should be injected by ConfigMaps, Secrets, external secret stores, or other controlled systems. That is similar to the practice of reading configuration from argv or a config file at startup rather than hardcoding it as a compile-time constant — the binary stays generic, and behavior changes without rebuilding.

Volumes as an externalized state

C++ engineers are used to thinking about memory lifetime. Stack variables disappear when a function returns. Heap objects live until released. Files persist beyond process lifetime. Shared memory may outlive one participant but not another. Containers are ephemeral. When a container dies, its writable layer is not a durable application state.

C++ / OS Idea	Kubernetes Idea
Stack variable	Container-local ephemeral state
Heap object owned by process	Runtime state inside a container
Temporary file	emptyDir volume
File on persistent disk	PersistentVolume
Claim to external storage	PersistentVolumeClaim
Shared mounted resource	Volume shared between containers

The key lesson is that Pod lifetime and data lifetime are different. A Pod may be destroyed and recreated. If payment-service requires a durable state, that state must be

externalized to a database, object store, persistent volume, or another storage system. A pointer to process memory is not a durable storage strategy; neither is a write to a container filesystem.

Namespace, RBAC, and NetworkPolicy as boundaries

Kubernetes Namespaces divide cluster resources into named scopes. The analogy with C++ namespaces is obvious but incomplete. C++ namespaces organize names. Kubernetes Namespaces also support access control, resource quotas, policy separation, and operational grouping. A better analogy is: C++ namespace plus build-module boundary plus administrative scope.

The danger is assuming that a Namespace is a hard security boundary by itself. It is not. Security also depends on RBAC, NetworkPolicy, admission control, pod security settings, secret handling, runtime configuration, and cluster design.

RBAC controls what users, service accounts, and workloads can do to Kubernetes resources. It resembles C++ access control only in the broadest sense. C++ private/protected/public is compile-time language enforcement. Kubernetes RBAC is runtime API authorization. A Pod can run under a ServiceAccount with permissions. Those permissions should be minimal.

NetworkPolicy moves encapsulation from code structure to network topology. It can declare that only frontend Pods may communicate with backend Pods on a given port. Without such a policy, many clusters allow broad communication by default depending on the networking implementation[viii]. That is like making every internal object globally reachable. With NetworkPolicy, you can explicitly express architectural boundaries.

Helm, Kustomize, CRDs, and Operators

Kubernetes manifests can become repetitive and environment-dependent. Helm and Kustomize are common tools for packaging or customizing YAML. Helm is often like a package manager and a templating system combined. Kustomize is more like declarative patching and overlay composition. The danger is familiar from C++ build systems: configuration complexity can become a system unto itself. A simple service with a monstrous chart is as hard to reason about as a simple program with a monstrous build system.

Custom Resource Definitions (CRDs) add new object kinds to the Kubernetes API: DatabaseCluster, Certificate, BackupPolicy, ModelDeployment, TracePipeline, and GPUWorkload. An Operator is a controller that watches custom resources and reconciles real infrastructure to match them. A GPUWorkload CRD, for example, might let you declare that a machine-learning inference service needs two A100 GPUs, a specific CUDA version, and a model checkpoint mounted from object storage — the Operator then handles node selection, driver validation, checkpoint download, and

health monitoring, rather than you manually orchestrating those steps through raw Pods and Jobs.

For C++ engineers, a CRD is like declaring a new type, and an Operator is like implementing all the lifecycle rules that type needs — construction, health monitoring, upgrade logic, and teardown — as a background process running in the cluster. The key distinction from a simple class: an Operator's "reconcile" is not a single method call. It is a continuous loop that watches for changes, computes the difference between the desired and actual states, applies corrective actions, handles partial failures, manages retries with backoff, and updates status. This is metaprogramming for Kubernetes: the CRD is the object model, the Operator is the domain runtime, and the cluster is the execution environment.

Kubernetes YAML as declarative object construction

C++ engineers may initially dislike Kubernetes YAML because it feels verbose and indirect. A better view is that YAML manifests are declarative object construction. A manifest describes an object, but creating that object in Kubernetes does not merely allocate memory. It creates a durable desired state in the API server. Controllers then act on that state.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-service
spec:
  replicas: 3
```

This is somewhat like aggregate initialization, but with a crucial difference: the object participates in an ongoing control system. A Kubernetes manifest may cause Pods and network routes to be created, volumes to be mounted, credentials to be exposed, and controllers to execute continuing behavior. YAML is not a passive configuration. It is an executable architecture in a declarative form — with the important caveat that the "execution" is interpretation by controllers, not direct imperative commands.

Part IV — Diagnostics at Cluster Scale

Kubernetes introduces a diagnostic abstraction layer that sits above ordinary OS-level and memory analysis. In traditional C++ debugging, you ask: Which thread crashed? Which heap object is corrupt? Which module owns this address? Which lock is blocking progress? In Kubernetes, those questions still matter — but they come second. The first questions are at the cluster level: which Pod owns this container? Which controller owns this Pod? Which node accepted the workload? Which probe declared it unhealthy? Which rollout created it? Which control-plane decision changed its fate?

These two levels are complementary, not competing. Cloud-level evidence narrows the search space until it points at a specific container, process, thread, or heap. At that point, traditional C++ tooling — logs, core dumps, debuggers, memory analysis — takes over. The process is not gone; it is embedded inside a larger orchestration topology. Good Kubernetes diagnosis means knowing which level to interrogate first.

The Cloud Analysis Patterns View

The previous sections used C++ analogies to make Kubernetes easier to understand. The Cloud Analysis Patterns view goes one step further. It says that Kubernetes is not merely a deployment platform for processes. It creates a new diagnostic space above the process, above the host, and above traditional memory analysis.

At the operating-system level, Kubernetes is still implemented using ordinary mechanisms: processes, namespaces, cgroups, filesystems, sockets, signals, images, and container runtimes. A containerized C++ service is still a process with threads, memory, file descriptors, dynamic libraries, logs, and the potential for crashes. Nothing mystical happens at this level. If payment-service dereferences a null pointer, leaks memory, mishandles SIGTERM, or fails because libstdc++.so is missing, the root cause may still belong to ordinary native-code diagnostics.

But Kubernetes adds another layer of meaning on top of that process. At the cloud-analysis level, we are no longer asking only which thread crashed, which heap object is corrupt, which module owns an address, or which frame explains the exception. We are also asking which Pod contained the process, which Deployment created the Pod, which ReplicaSet[ix] version it belonged to, which Node accepted it, which Service selected it, which readiness probe excluded it from traffic, which ConfigMap or Secret shaped its environment, which NetworkPolicy constrained its communication, and which controller attempted to repair the situation.

This is the key shift: cloud diagnostics is not merely memory diagnostics performed inside containers. It is diagnostics of orchestration structures.

A traditional C++ postmortem often starts with a local artifact: a core dump, a stack trace, a log file, a crash address, a failed assertion, or a corrupted object. Kubernetes postmortem analysis often starts with a relational artifact: a Pod state, an Event sequence, a controller decision, a rollout transition, a Service selector, an EndpointSlice[x], a probe result, a scheduling failure, or a volume-binding problem. The evidence is not only inside the process. It is distributed across the cluster object graph.

For this reason, Kubernetes objects should be treated as diagnostic relations rather than just deployment resources. A Pod relates containers to a node, a network namespace, volumes, labels, probes, and ownership metadata. A Service relates stable naming to a changing set of selected Pods. A Deployment relates desired replica count to ReplicaSets and rollout history. A controller relates declared intent to corrective

action. An Operator relates a domain-specific object to a set of lower-level Kubernetes resources and external effects.

This gives a more precise version of the analogy introduced earlier:

```
Traditional C++ diagnostics:
    process -> thread -> stack -> frame -> object -> pointer

Kubernetes diagnostics:
    cluster -> node -> Pod -> container -> process -> thread -> stack

Cloud Analysis Patterns:
    controller relation -> ownership relation -> selection relation
    -> scheduling relation -> readiness relation -> traffic relation
    -> container/process evidence
```

The diagnostic direction is often top-down. We may begin with a stalled rollout, a missing endpoint, or a failing Service. Then we descend through Deployment, ReplicaSet, Pod, container, process, logs, and eventually native debugging artifacts. But the direction can also be bottom-up. A process crash may explain a container restart; the restart may explain a Pod not becoming Ready; the NotReady Pod may explain a missing endpoint; the missing endpoint may explain a failed rollout; the failed rollout may explain a customer-visible outage.

Consider CrashLoopBackOff. At the C++ level, it may be caused by a segmentation fault, a failed assertion, a missing library, an invalid configuration, or an unhandled exception. At the Kubernetes level, however, CrashLoopBackOff is not simply “the program crashed.” It is a cloud-level symptom involving container exit status, kubelet restart behavior, restart policy, backoff timing, Pod status, Events[xi], logs, probes, resource limits, and controller expectations. The process failure is embedded inside an orchestration pattern.

The same is true of a rollout failure. It may look like a Deployment problem, but the real issue may be a configuration-interface mismatch, a readiness probe contract violation, a Service selector error, a schema incompatibility, an image pull failure[xii], or a resource constraint. A missing endpoint is not merely a networking issue. It may be a label problem, a readiness problem, a Pod lifecycle problem, a controller ownership problem, or an application initialization problem.

Cloud Analysis Patterns therefore preserve traditional diagnostics while changing their position in the hierarchy. Memory analysis, stack analysis, log analysis, and native debugging do not disappear. They become lower-level instruments inside a larger cloud-native reconstruction. The diagnostician first identifies the relevant orchestration relation, then descends into the process only when the cloud-level evidence points there.

For C++ engineers, this is the strongest form of the Kubernetes analogy. The familiar world of processes, threads, heaps, stacks, resources, ownership, invariants, and

failures remains intact. But it is now embedded inside a larger topology of Pods, Services, controllers, policies, rollouts, probes, nodes, and Events. Kubernetes is not only where the C++ process runs. It is the relational diagnostic space in which the process becomes observable, replaceable, selectable, restartable, routable, and accountable.

This is why Kubernetes rewards systems thinking. The engineer who can reason only about YAML sees resources. The engineer who can reason only about C++ sees processes. The Kubernetes diagnostician must see both: the native execution space below and the cloud orchestration space above.

Probes as assertions and health contracts

Kubernetes uses probes to determine whether containers are alive, ready, or successfully started. For a C++ engineer, probes are like runtime assertions exposed to the orchestration system, but they are also operational contracts.

Probe	Question
Startup probe	Has the application finished starting?
Liveness probe	Is the container still healthy, or should it be restarted?
Readiness probe	Is the container ready to receive traffic?

The distinction between liveness and readiness is critical. A process may be alive but not ready. payment-service may have started its event loop but not yet loaded the configuration, connected to the database, warmed the caches, or replayed the log. If readiness is wrong, Kubernetes may send traffic too early. If liveness is wrong, Kubernetes may kill a process that is merely slow, overloaded, or recovering. This is similar to writing a bad assertion in C++: the wrong invariant can turn a recoverable condition into a crash.

CrashLoopBackOff as repeated process failure under supervision

C++ engineers understand crashes: segmentation faults, aborts, unhandled exceptions, invalid memory accesses, failed assertions, missing shared libraries, bad configurations, and incompatible CPU instructions. In Kubernetes, when a container repeatedly starts and fails, the Pod may enter a CrashLoopBackOff state. This means the container exited or crashed, Kubernetes restarted it, and repeated failures caused increasing backoff delays[xiii].

```
while true; do
    ./payment-service
    sleep backoff
done
```

The Kubernetes status is not the root cause. It is the symptom pattern. The C++ diagnostic questions remain valuable: what was the entry point, did dynamic loading succeed, were required files present, were environment variables valid, was the process killed by memory pressure, did it receive a signal, did it fail an assertion, and what did previous logs say? Kubernetes changes the diagnostic surface, but not the underlying reality that a process failed or was terminated.

kubectl as a debugger, inspector, and control console

C++ engineers use debuggers, profilers, tracing tools, logs, core dumps, disassemblers, and system tools. Kubernetes has its own inspection surface, and kubectl is the most common entry point.

Native Debugging	Kubernetes Inspection
ps	kubectl get pods
Process details	kubectl describe pod
stdout/stderr	kubectl logs
Previous process output	kubectl logs --previous
Attach shell	kubectl exec
Debugger attach	Ephemeral debug container or exec
System event log	Kubernetes Events

```
kubectl get pods
kubectl describe pod payments-abc123
kubectl logs payments-abc123
kubectl logs payments-abc123 --previous
kubectl exec -it payments-abc123 -- /bin/sh
```

The --previous option is particularly useful for crash diagnosis, since the current container may have been restarted. The previous container instance may contain the relevant crash logs. Kubernetes debugging often begins by reconstructing process history: Was the Pod scheduled? Did the image pull? Did the container start? Did the process run? Did it crash? Was it killed? Did probes fail? Was traffic sent to it? Was the configuration correct?

One place C++ engineers often get stuck: correlating logs across multiple replicas. Unlike a single-process crash where you have one core dump, a Kubernetes incident may involve one Pod crashing while two others continue. Use kubectl logs with the -l selector flag to gather logs across all matching Pods simultaneously[xiv], and check Kubernetes Events (kubectl get events --sort-by=.lastTimestamp) to reconstruct the sequence of scheduling decisions, probe failures, and restarts.

A diagnostic mini-scenario: the rollout that did not fail where it seemed to fail

Consider payment-service version 1.1. The Deployment is updated, and the rollout stalls. At first glance, the problem appears to be a Kubernetes rollout failure.

```
kubectl rollout status deployment/payment-service
# Waiting for deployment "payment-service" rollout to finish:
# 1 out of 3 new replicas have been updated...

kubectl get pods
# payment-service-1-0-abc   Running   1/1   Ready
# payment-service-1-0-def   Running   1/1   Ready
# payment-service-1-1-xyz   Running   0/1   NotReady
```

The new Pod is running but not ready. That distinction matters. The process did not necessarily crash. The container started, but the readiness contract failed. describe shows readiness probe failures. logs show that payment-service version 1.1 expects DB_DSN, while the Deployment still injects DATABASE_URL from the old ConfigMap or Secret.

```
kubectl describe pod payment-service-1-1-xyz
# Readiness probe failed: HTTP probe failed with status code: 503

kubectl logs payment-service-1-1-xyz
# payment-service: started
# config: DB_DSN is empty
# database: not initialized
# readiness: false
```

The diagnosis spans several levels. At the C++ level, the process started and executed correctly enough to serve a health endpoint. At the container level, an expected environment variable was missing. At the Pod level, readiness failed. At the Service level, the Pod was removed from the ready endpoint set. At the Deployment level, the rollout stalled because the new replicas did not become available. The apparent Kubernetes problem is a configuration-interface incompatibility exposed by orchestration.

Level	Evidence	Interpretation
C++ process	Process starts and logs the configuration problem	No immediate crash; semantic initialization failure
Container	Environment variable mismatch	Image and manifest contract diverged
Pod	Ready = false	The local runtime unit is not fit for traffic
Service	Endpoint not ready	Traffic should not be routed to this Pod
Deployment	Rollout stalls	Controller refuses to advance because availability invariant is not met

This is the kind of diagnostic situation where C++ instincts and Kubernetes knowledge reinforce each other. The binary may be correct. The YAML may be syntactically correct. The failure lives in the contract between them.

Common traps for C++ engineers

The analogies are useful precisely because they are not perfect. The following traps are ranked roughly by how quickly they will hurt you.

Will hurt you immediately:

- Assuming that if the process starts, the service is available. Kubernetes separates process liveness from service readiness. Traffic routing depends on readiness probes, not process existence.
- Assuming that a successful manifest apply means the workload is running. It only means the desired state was accepted. The actual workload may still be pending, pulling an image, failing probes, or blocked by scheduling constraints.
- Assuming that local filesystem writes are durable. Container-local state is disposable. A Pod restart loses anything written to the container's writable layer.

Will bite you eventually:

- Assuming that a crash is always an application bug. The process may have been killed by memory limits, probe failure, node pressure, eviction, missing volumes, or bad rollout behavior.
- Assuming that rolling updates are only a deployment problem. They are also compatibility tests across versions, schemas, protocols, and configuration contracts.
- Assuming sidecars are just threads. They are separate containerized processes with their own images, filesystems, lifecycles, and failure modes.

Architectural mistakes:

- Assuming that a Namespace is a hard security boundary. It is a scope; isolation requires additional policy and configuration.
- Assuming that a Service validates protocol correctness. It provides network indirection; it does not enforce semantic compatibility.

Where the analogies break

A Pod is not a process group in the full POSIX sense: containers do not share a PID namespace by default, signal delivery is per-container, sidecars may have distinct lifecycle behavior, and newer or feature-gated Kubernetes capabilities allow more fine-grained container restart behavior. A Service is not a type-safe interface. A Namespace is not a security boundary by itself. A Deployment is not deterministic RAII. A controller is not a simple function call. A container image is not merely an executable. A cluster is not one machine. A manifest is not a passive configuration.

The biggest difference is determinism. C++ often gives the illusion, and sometimes the reality, of local deterministic execution. Kubernetes operates in a distributed environment with eventual convergence. Things happen asynchronously. Controllers observe, compare, and act. State changes may be delayed. Objects may exist while their consequences are still pending.

In C++, if construction succeeds, the object exists in memory. In Kubernetes, if object creation succeeds, the desired state exists in the API server. The actual running workload may still be pending, pulling an image, waiting for a volume, failing probes, blocked by scheduling constraints, or being rejected by admission policy. That difference is fundamental. Kubernetes is not imperative execution. It is declarative convergence.

Conclusion: systems thinking at the cluster scale

Kubernetes does not make C++ systems thinking obsolete. It changes its scale. The process is still there, with its heap, threads, file descriptors, signals, libraries, logs, and crashes. But it is now embedded inside a larger topology of Pods, Services, controllers, rollouts, probes, policies, volumes, and nodes.

For C++ engineers, the right question is not “How do I learn YAML?” The better question is: how are the systems concerns I already understand represented at the cluster scale? Kubernetes is one answer: a distributed runtime where ownership becomes controller ownership, construction becomes desired state, failure becomes reconciliation, interfaces become Services, assertions become probes, and debugging becomes reconstruction across processes, containers, Pods, nodes, and clusters.

The analogy should not be forced. A Pod is not a process group, and a Service is not a type-safe interface. But the analogy is productive because it gives C++ engineers a disciplined entry point. Kubernetes is not a departure from systems thinking. It is systems thinking at the cluster scale.

References

Kubernetes Documentation: Pods — https://kubernetes.io/docs/concepts/workloads/pods/
Kubernetes Documentation: Sidecar Containers — https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/
Kubernetes Documentation: Deployments — https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
Kubernetes Documentation: Services — https://kubernetes.io/docs/concepts/services-networking/service/
Kubernetes Documentation: Controllers — https://kubernetes.io/docs/concepts/architecture/controller/
Kubernetes Documentation: Resource Management for Pods and Containers — https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
Kubernetes Documentation: StatefulSets — https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
Kubernetes Documentation: Network Policies — https://kubernetes.io/docs/concepts/services-networking/network-policies/
Cloud Analysis Patterns — https://www.dumpanalysis.org/cloud-analysis-patterns

Quick Reference: The Analogy Map

The following table summarises all the analogies covered in this article.

C++ / Systems Concept	Kubernetes Concept	Analogy
Executable	Container image	Packaged runnable artifact
Process	Container	Running instance of image
Process group	Pod	Colocated containers sharing context
Object owner	Deployment	Maintains replica lifetime
Runtime invariant	Desired state	State controllers try to preserve
Background repair loop	Controller reconciliation	Converges actual to desired state
Allocator	Scheduler	Places workloads on available resources
Machine	Node	Execution host
Namespace/module scope	Namespace	Naming and administrative scope
Type traits/tags	Labels	Metadata for grouping and selection
Query over traits	Selector	Chooses matching objects
Interface	Service	Stable access to changing implementations
Public API facade	Ingress/Gateway	External route into internal services
Runtime config file	ConfigMap	Non-sensitive configuration
Credential store	Secret	Sensitive runtime data with policy requirements
External storage	Volume	State outside container lifetime
std::array with identity	StatefulSet	Stable indexed replicas
Per-host daemon	DaemonSet	One Pod per node
Command-line utility	Job	Run-to-completion task
Scheduled command	CronJob	Periodic task
Access modifier	RBAC	Runtime authorization
Encapsulation boundary	NetworkPolicy	Allowed communication
Custom class/type	CRD	New Kubernetes object kind
Domain runtime / lifecycle manager	Operator	Controller for a custom resource

Debugger/system tools	kubectl	Inspection and control
Assertions/contracts	Probes	Health and readiness contracts
Crash restart loop	CrashLoopBackOff	Repeated process failure under supervision

[i] PID 1 is the first process started inside a container, and it has responsibilities that most C++ engineers never think about because in a normal Linux system, init (or systemd) handles them automatically.

There are two main issues here:

Signal forwarding. When Kubernetes wants to stop a container gracefully, it sends SIGTERM to PID 1. If your payment-service binary is PID 1 (which it is if you use CMD ["./payment-service"] directly in the Dockerfile), it must handle SIGTERM itself and shut down cleanly. Most C++ services do handle SIGTERM fine. The problem arises when you launch your binary via a shell script — the shell becomes PID 1, receives SIGTERM, but doesn't forward it to the child process. Your service never gets the signal, Kubernetes waits for the grace period, then sends SIGKILL. You get a hard kill instead of a graceful shutdown, which can mean unflushed logs, incomplete requests, or corrupted state.

Zombie reaping. PID 1 is also responsible for reaping zombie processes — children that have exited but whose exit status hasn't been collected with wait(). In a normal system, init does this automatically. If your C++ binary is PID 1 and it spawns child processes (for diagnostics, forking workers, etc.) that exit, those become zombies. Most C++ services don't spawn children so this rarely bites, but it's the kind of thing that causes mysterious slow memory leaks in long-running containers.

In local Docker usage, docker run --init inserts a small init process. In Kubernetes, the more portable pattern is to build a minimal init such as tini or dumb-init into the image entrypoint, or ensure that the process running as PID 1 handles signal forwarding and child reaping correctly.

This is not unique to C++, but bare native binaries and shell-wrapped entrypoints make the issue especially visible. The safest rule is language-neutral: know what runs as PID 1, ensure signals are handled or forwarded, and use a minimal init process when child reaping or forwarding is needed.

[ii] They do not share the PID namespace by default (though this is configurable via shareProcessNamespace: true).

[iii] A directly created Pod can have its containers restarted by the kubelet while the node is healthy, depending on restartPolicy. What it does not get is controller-level replacement. If the node fails or the Pod disappears, no Deployment, ReplicaSet, Job, or StatefulSet exists to create a replacement Pod on another node.

[iv] Kubernetes does not queue traffic for a Service until a ready backend appears. If a Service has no usable ready endpoints, requests fail according to the dataplane and caller path: they may time out, be refused, be reset, or produce an upstream error.

[v] Traditionally, requests and limits are specified per container, and the scheduler reasons about the Pod as the schedulable unit by considering the aggregate resource needs of its containers. Newer Kubernetes versions also support Pod-level resource budgets for some resources, but the practical scheduling lesson remains the same: placement is decided for the Pod as a whole, not for independent containers.

[vi] A Deployment is like std::vector<std::unique_ptr<Worker>> where every element is interchangeable — you care about the count, not the identity, and any element can be destroyed and reconstructed without affecting the others. A StatefulSet is more like std::array with named slots: node[0], node[1], and node[2]

have stable ordinal identities, predictable DNS names, and individually bound PersistentVolumes — destroying and recreating node[1] returns it to the same identity and storage, not an anonymous replacement.

[vii] DaemonSets respect taints/tolerations and node selectors. On a fresh cluster with control-plane taints, DaemonSet Pods don't run on control-plane nodes by default unless explicitly tolerated.

[viii] A NetworkPolicy object is only meaningful if the cluster uses a network plugin that enforces NetworkPolicy; otherwise the object may exist without enforcing the intended isolation.

[ix] A ReplicaSet is the object that actually owns and maintains a set of identical Pod replicas. It's the thing doing the direct work of "keep N copies of this Pod running."

You rarely interact with ReplicaSets directly. Instead the relationship is:

```
Deployment
  └── ReplicaSet (version 1.0)
  └── ReplicaSet (version 1.1)  ← new one created on rollout
      └── Pod
      └── Pod
      └── Pod
```

Why it exists as a separate layer: when you update a Deployment (say, bumping the image from 1.0 to 1.1), Kubernetes doesn't modify the existing ReplicaSet. It creates a new ReplicaSet for the new version and gradually scales it up while scaling the old one down. The old ReplicaSet usually remains available for rollback, subject to the Deployment’s revision history and cleanup settings. Rolling back means scaling the old one back up and the new one back down.

The C++ analogy: if a Deployment is the std::vector that declares "I want 3 workers," a ReplicaSet is closer to the actual heap allocation backing it at a specific version. The Deployment is the stable owner you interact with; the ReplicaSet is the versioned implementation detail underneath. You don't new a ReplicaSet directly any more than you manually manage the internal buffer of a vector.

In practice the only time you look at ReplicaSets directly is during rollout debugging — kubectl get replicasets shows you the history of versions and their current/desired replica counts, which tells you exactly where a stalled rollout is stuck.

[x] EndpointSlices track the backend endpoints associated with a Service and include conditions such as ready, serving, and terminating. For normal Service routing, Pods that are not Ready should not receive traffic, but diagnostically it is better to think in terms of endpoint conditions rather than only “present” or “removed.”

When you create a Service with a label selector, Kubernetes continuously watches for Pods matching that selector and maintains EndpointSlices that list the healthy, ready Pod endpoints. When your payment-service Pod becomes Ready, its IP and port get added to the EndpointSlice for the Service. When it dies, fails readiness, or begins terminating, its EndpointSlice entry may be removed or updated with conditions such as ready, serving, and terminating; for normal Service routing, the important point is that not-ready endpoints should not receive ordinary traffic. The kube-proxy (or whatever networking layer your cluster uses) watches EndpointSlices and updates routing rules accordingly — that's how traffic actually gets to the right Pods.

Why "Slice" and not just "Endpoints": the original Endpoints object predates EndpointSlices and stored all backends for a Service in a single object. At scale — say, a Service with 500 Pod replicas — every time one Pod changed state, Kubernetes had to rewrite and redistribute that entire object to every node. EndpointSlices shard the backends into smaller chunks (default max 100 entries each), so a single Pod

change only touches one slice, not the whole set. It's the same reason you'd shard a large data structure rather than copying it wholesale on every update.

The C++ analogy: an EndpointSlice is roughly like a routing table or a watched list of live object pointers behind an interface. The Service is the stable interface name; the EndpointSlice is the runtime dispatch table that maps that name to currently valid targets. When a Pod restarts and gets a new IP, the EndpointSlice is updated — the interface stays stable, the backing implementation changes underneath it.

When you encounter it in practice: usually when debugging missing traffic. If a Service exists but requests aren't reaching Pods, kubectl get endpointslices -l kubernetes.io/service-name=payment-service shows you whether any ready endpoints are registered at all. An empty EndpointSlice means either no Pods matched the selector, or none passed their readiness probe — and that distinction tells you exactly where to look next.

[xi] Kubernetes Events are structured log entries that record things that happened to objects in the cluster — scheduler decisions, kubelet actions, controller reconciliations, probe failures, image pulls, volume mounts, and so on. Every significant state change or error that Kubernetes itself produces gets recorded as an Event attached to the relevant object.

What they look like:

```
kubectl get events --sort-by=.lastTimestamp -n default

LAST SEEN   TYPE      REASON              OBJECT                          MESSAGE
2m          Normal    Scheduled           pod/payment-service-abc         Assigned to node worker-2
2m          Normal    Pulling             pod/payment-service-abc         Pulling image "example/payment-service:1.1"
1m          Normal    Pulled              pod/payment-service-abc         Successfully pulled image
1m          Normal    Created             pod/payment-service-abc         Created container payment-service
1m          Normal    Started             pod/payment-service-abc         Started container payment-service
30s         Warning   Unhealthy           pod/payment-service-abc         Readiness probe failed: HTTP 503
10s         Warning   BackOff             pod/payment-service-abc         Back-off restarting failed container
```

The C++ analogy: Events are closest to a structured system log or a flight recorder — similar to what you'd get from ETW on Windows or from a tracing framework like dtrace. They are not application logs (those come from your process's stdout/stderr via kubectl logs). They are the runtime's own account of what it did and why, one level above your process.

Three things that make Events distinctive:

First, they are ephemeral. By default, Kubernetes retains Events for only one hour in many cluster configurations. This is the most common trap: a Pod crashed and restarted, you investigate twenty minutes later, and the Events that would have told you exactly what happened are gone. In production clusters, it's worth shipping Events to a persistent store.

Second, they have a Type field: Normal or Warning. Warnings are the ones to filter for first during an incident.

Third, they are scoped to an object but readable cluster-wide. kubectl describe pod payment-service-abc shows Events for that Pod inline. kubectl get events shows everything. In a large cluster, filtering by namespace and sorting by time is essential.

Why they matter for C++ engineers specifically: the diagnostic pattern in Kubernetes is often to start with Events before touching logs. Events answer the orchestration-level questions — was the Pod scheduled? Did the image pull? Did the probe fail? — before you descend into the process-level questions that kubectl logs and core dumps answer. They are the top of the diagnostic stack, not the bottom.

[xii] Image pull is the step where the container runtime on a node downloads the container image from a registry before it can start the container.

When Kubernetes schedules a Pod onto a node, the kubelet instructs the container runtime (containerd, CRI-O, etc.) to ensure the required image is present locally. If it isn't cached on that node, the runtime fetches it from wherever the image is stored — Docker Hub, a private registry, a cloud provider registry like ECR or GCR.

The sequence:

```
Pod scheduled onto node
  -> kubelet checks local image cache
  -> image not present (or pull policy requires fresh fetch)
  -> runtime pulls image layers from registry
  -> layers cached on node
  -> container started from image
```

The C++ analogy: it's roughly equivalent to the dynamic linker resolving and loading shared libraries at process startup — except it happens at the node level before the process even starts, and the "library" is an entire filesystem snapshot rather than a single .so. A missing image is like a missing shared library: the process never gets to run. The error you see is ErrImagePull or ImagePullBackOff in Pod status, which is the Kubernetes equivalent of error while loading shared libraries.

The practical failure modes C++ engineers hit:

ImagePullBackOff means the pull failed repeatedly and Kubernetes is backing off with increasing delays before retrying. Common causes are a wrong image name or tag, the image doesn't exist, the registry is unreachable, or missing pull credentials for a private registry.

Image pull policy controls when the pull happens. IfNotPresent uses the cached image if it exists on the node — fast, but you can end up running a stale image if you pushed a new version under the same tag. Always forces a fresh check on every Pod start — safer for mutable tags like latest, slower. Never assumes the image is already on the node and fails if it isn't.

The layer caching detail matters for C++ engineers building images: container images are built in layers, and layers are cached independently on each node. If your C++ binary is in the top layer and your base OS and dependencies are in lower layers, only the top layer needs to be pulled on updates. A poorly structured Dockerfile that puts the binary in an early layer invalidates all subsequent layers on every build, making pulls unnecessarily large. The standard pattern is: `OS base -> system dependencies -> application dependencies -> compiled binary`, so that only the last layer changes on most rebuilds.

[xiii] The actual backoff sequence in Kubernetes is 10s, 20s, 40s, …, capped at 300 seconds, and resets after the container runs successfully for 10 minutes.

[xiv] kubectl logs -l collects logs from Pods that currently match the selector and still exist. If the crashing Pod has already been deleted and replaced, selector-based log collection will not recover its old logs. For multi-container Pods, specify -c or use --all-containers where appropriate.