Architecture Paper



- What Is Aurae?
  - Where did Aurae Come From?
- FAQ
  - Will Kubernetes work with Aurae?
  - Does Aurae replace Kubernetes?
  - Does Aurae replace systemd?
  - Does Aurae use etcd?
- Simple Beginnings
- Scaling Aurae
- Peer Networking Model
- Peer to Peer Mesh
  - Service Discovery
  - Control Systems (Control Plane)
  - Distributed Hash Table (DHT)
  - Aurae Daemon Identity and Authentication Tani Aurae
  - Network Devices are Services
  - Service Transit Systems (Data Planes)
  - Joining a Node to a Mesh
- The Aurae Standard Library
  - Aurae Language
  - Manifests
- Appendix
  - Cluster vs Mesh
  - Decision to stay away from BitTorrent infrastructure

What Is Aurae?

Aurae is an opinionated systems kernel designed to be a building block for higher level distributed systems such as Kubernetes. Aurae intends to simplify and secure the relationship between a Node and a distributed system, as well as provide powerful networking primitives to higher order systems in the stack.

A primary goal of Aurae is to replace systemd. We also aim to replace the runtime node components in a modern distributed system such as Kubernetes.

Aurae calls out a standard library and a Turing complete scripting language on which platform teams can develop and compose their platforms in an opinionated and scalable way. Our hope is that distributed systems such as Kubernetes and modern service mesh systems will find value in building on top of Aurae.

Aurae is the Turing complete platform language designed for platform teams.

Auraed is the core daemon that supports it.

Where did Aurae Come From?

We believe that there is untapped opportunity in how we manage nodes in distributed systems. Specifically, we believe that better multi-tenant building blocks at the node level will unlock more effective platform abstractions (such as the Kubernetes control plane) on top.

The project is a result of a simplification of the core needs of a modern platform infrastructure team. Aurae attempts to replace systemd and the lower level Kubernetes components that bring a traditional Kubernetes environment to life.

Aurae was originally started by Kris Nóva. The project draws inspiration from Plan 9, Kubernetes, and the COSI project. The primary motivation for the project was to follow the dream that distributed systems kernels could adopt a broader scope while also undergoing simplification.

Core engineers to the project in its early stages include Duffie Coolie, and Tani Aura.

Works that influenced and validated the project include

- Sketch of the Biggest Idea in Software Architecture
- A New Kubernetes from the Ground Up
- Kubernetes StatefulSets are Broken

Authors: Kris Nóva

FAQ

Will Kubernetes work with Aurae?

Yes. We intend to simplify a layer of the stack. The Aurae API will solve many of the same problems as Kubernetes, and there is no reason a translation between Kubernetes objects and Aurae API calls cannot be maintained. In fact – the project calls out the ability to have a transparent Kubernetes deployment running on top of Aurae as one of its goals.

Does Aurae replace Kubernetes?

It depends – but mostly no. Think of Aurae as systemd and the kubelet wrapped up into a single system that takes networking, storage, identity, and runtime into scope. Instead of managing an entire Linux system, kernel, systemd services, container runtimes, CNI, CSI, and etc underneath Kubernetes, you can just run Aurae on the node instead. The Aurae APIs will enable the same functionality that is otherwise available on a traditional Kubernetes node. Aurae aims to allow an engineer to leverage the Aurae mechanics to support higher level orchestration systems such as the Kubernetes API and control plane. Aurae goes after the node, not the cluster.

Does Aurae replace systemd?

Aurae intends to replace systemd and the kubelet in a single fell swoop. Will it be successful in its mission? All we can do now is pray.

Does Aurae use etcd?

No.
Aurae does not have a centralized source of truth like a Kubernetes cluster does. However Aurae does persist configuration at the node level. Each node in a mesh is responsible for maintaining its own source of truth. Additionally each node provides functionality that allows a system on one node to mutate a system on another node.

We use SQLite for our primary data store. The database runs on each Node instead of a centralized model.

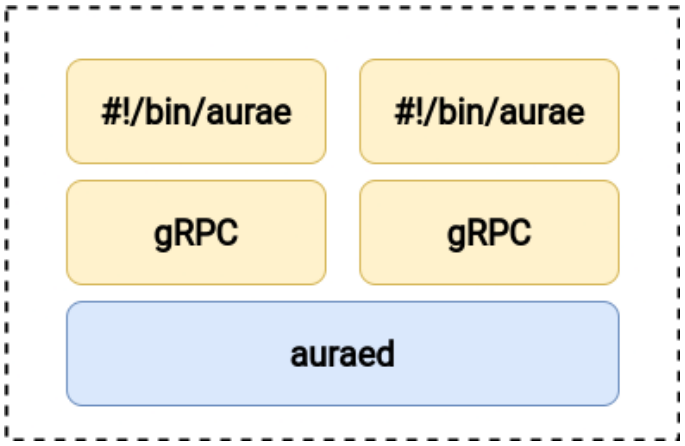
Also for consideration SpiceDB which has authorization and identity baked in at the database level.

Authors: Kris Nóva

Simple Beginnings

The simplest model for Aurae is running multiple tenants on a single system.

Each tenant (in this example) leverages the Aurae language over mTLS gRPC over a Unix domain socket. Each tenant has unique certificate material loaded at runtime. Each tenant executes against the same core daemon. Each tenant is reasoned about independently by the daemon based on the tenant's identity at runtime.

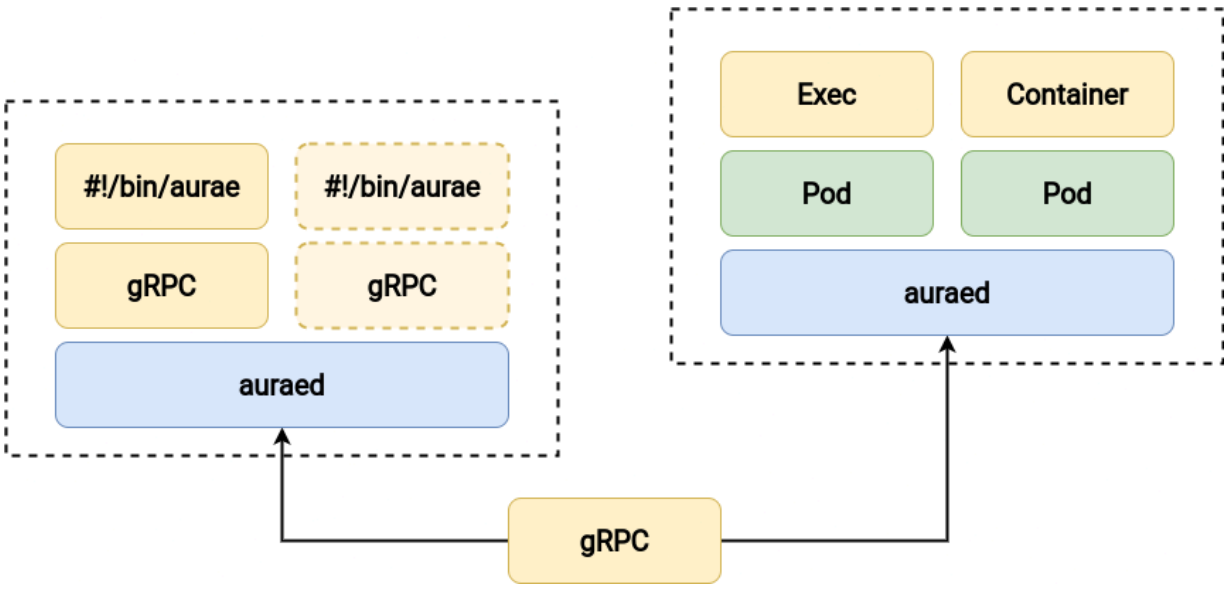


Containers run in pods. Pods run in MicroVM isolation sandboxes. Sandboxes are the new namespace. They all run on a single piece of hardware.

Scaling Aurae

Aurae is intended to grow organically with the needs of a business. Starting with Aurae is simple as the recommendation is to always start with a single node. Add a second node when you have reached a critical size on the first node. And so on.

Aurae nodes build and maintain a peer-to-peer mesh at runtime.



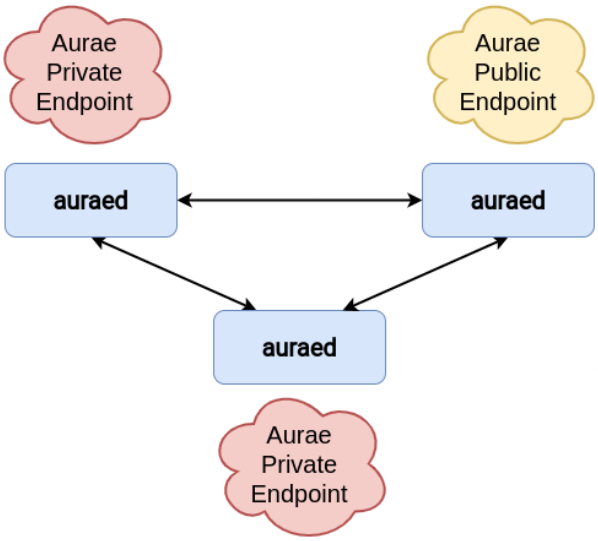
Authors: Kris Nóva

Peer Networking Model

Auraed will listen on a Unix domain socket by default. Auraed will manage network devices for the system itself, as well as guests.

Auraed will provide service discovery and a lookup mechanism for other nodes in the mesh.

Each connection from a single node to another will be a direct point to point connection.



Peer to Peer Mesh

Aurae is designed to work in a decentralized way. Work is scheduled directly where it runs, or work is not scheduled at all. The networking model is designed to be as flat as possible. Aurae nodes navigate the mesh via calculating Hamiltonian paths at runtime.

Aurae will also call out a node registry as a supported service in the future.

Service Discovery

Aurae improves DNS and addresses the decentralization problem by calling out a simple routing syntax for the nodes to follow.

@service@node@domain

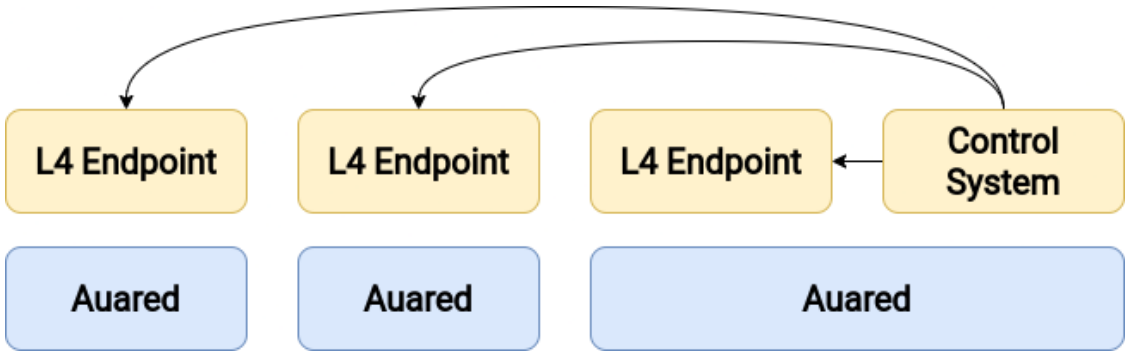
For example, if a user wanted to route a packet to a service, they would need to know the service name, the node name, and the domain name of the intended destination. For example, routing to my blog running on a node name “alice” would look like this.

@blog@alice@nivenly.com
@blog@@nivenly.com*
@blog@nivenly.com

Authors: Kris Nóva

Control Systems (Control Plane)

Aurae calls mesh management and higher order decision making out of scope. However the project will likely end up maintaining one or more higher order services that will reason about **where** to send various messages and instructions in a mesh. The control system will be “node aware” and will use the awareness to make scheduling, and routing decisions.



Distributed Hash Table (DHT)

Aurae leverages an internally scoped DHT for service discovery. The paradigm for a DHT resembles that of public DNS. Each Aurae node must be hydrated with a routable address in order to begin finding other nodes in the mesh.

The DHT hydration paradigm is no different than defining 8.8.8.8 in resolv.conf.

A small public facing service can be scheduled on an Aurae node to begin serving as the initial bootstrapping hop.

Aurae Daemon Identity and Authentication *Tani Aurae*

Every Aurae daemon will have a cryptographic identity based on a combination of Public Key Infrastructure (PKI) Certificates and SSH keys. An admin may connect and control an Aurae Daemon using an ‘SSH Certificate’ that is a combination of SSH keys and a signed time-limited certificate. The Aurae Daemon itself also receives an SSH Certificate that it uses to authenticate itself to incoming SSH connections, and to authenticate itself with other Aurae Daemons.

For production environments scaling beyond one one Aurae Daemon, the CA should be configured before the Aurae Daemon starts. A plugin model will be provided to support various key management solutions such as Cloud KMS or proprietary solutions.

An Aurae Daemon may also start in detached mode, where it is not federating with other systems. In detached mode, authorized SSH keys are configured to control who or what is

allowed to bootstrap the system. The Aurae Daemon will create a private CA, sign SSH keys, and return the SSH certificate to the user.

This approach should be flexible enough for a developer to get started quickly with safe defaults. For major enterprises, the PKI may be backed by KMS, HSM, or other devices, providing full control over the attestation and signing process.

Authentication between systems with different root CAs will require a mechanism to share the CA across boundaries. This will allow for arbitrary federation of identities between organizations.

There is also an opportunity to federate with cloud infrastructure identities using OIDC Connect that will be investigated.

Authorization is maintained separately from Authentication and consumes Aurae's cryptographic identity.

Network Devices are Services

The endpoints model is what gives Aurae the ability to power large service mesh topologies seen with projects like Istio, DAPR, and Linkerd. Instead of positioning a sidecar next to application, Aurae will instead deploy endpoints onto the node. These endpoints provide powerful networking functionality such as service discovery, NAT translation, Proxy routing, and name service resolution.

Aurae will ship with a set of opinionated flat networking endpoints by default, however more advanced networking topology will be possible simply by implementing the endpoint interface on the system.

Service Transit Systems (Data Planes)

Aurae will need to identify a way for services to communicate within the mesh. There are two networking layers that will need to be discussed.

Host Mesh Network	**Service Mesh Network**
Composed of node-aware network endpoints that are connected together. The network endpoints that form the host mesh run directly on a node and network proxy traffic to/from the local Aurae daemon. The host mesh network will be end-to-end	Pod level networking managed by injecting interfaces directly into pods and sandboxes. The possibility to leverage existing CNI networking toolchains here is in scope.

Authors: Kris Nóva

encrypted by default.	

Joining a Node to a Mesh

The minimal requirements to join one Aurae instance to another is a communication bridge and node awareness. In other words a public DHT, public DNS, or hard coded network addresses will need to be identified for a specific mesh. The more known routable nodes in a mesh, the more resilient the joining process will be.

It will be possible to join a node through a control system as the control system will be aware of every node in the mesh. Thus the only awareness a node will need, is the root control system.

Authors: Kris Nóva

The Aurae Standard Library

Aurae is built on the concept of subsystems, similar to Linux subsystems or Kubernetes resource groups.

Subsystem	Description	Examples
Runtime	Stateless executive subsystem for direct interaction with a system's runtime resources such as Linux processes, container runtimes, microVM hypervisors, process management, etc	runtime.Run(myPod) runtime.Run(myDaemon) runtime.Stop(myPod.name)
Schedule	Higher level stateful wrapper system for Runtime. Here is where systemd unit files, and Kubernetes manifests will become relevant. This subsystem is responsible for scheduling runtime events under various criteria.	schedule.Cron(myPod, "* * *") schedule.Now(myPod) schedule.Pin(myPod, node) schedule.Lax(myPod)
Secrets	Secrets should never enter a codebase. The goal is easy to do the right thing with secrets.	myPod.Env("user", "nova") myPod.Env("pass", secrets.Get("nova"))
Identity	Identity is a wrapper subsystem that brings certificate management, authorization (authz) and IAM identity as low as possible in the stack. Auditing and identity should be easy to set up and manage by default.	nova = identity.User("nova) nova.Allow(runtime) nova.Allow(runtime.run) nova.Allow(runtime.Stop) nova.Deny(schedule.Cron)
Observe	Aurae will capture all stdout and stderr on a system and manage it for the daemon. Observe is how this data, and other data is accessed.	observe.Stdout(myPod) observe.Stderr(myPod) observe.Stdevent(myPod) observe.Metrics(myPod) observe.Stdout(myPod).withContext(ctx)
Route	Routing is a network abstraction that abstracts most of a networking stack away from the daemon. Here is where endpoints take over with service to service routing. The default routing module will enable transparent end-to-end encryption using the same X509 certificates the client connects to the daemon with.	route.Open(myPod, "@foo@nivenly") route.Open(myPod, "@baz@nivenly")
Batch	The batch system is a way of mutating large groups of Aurae objects at runtime. This feature is a core primitive of Aurae and will replace systems such as Helm for managing YAML.	myPod1.Name = "nova" myPod2.Name = "alice" myPod3.Name = "emma" batch.Name("overwrite", ["myPod1", "myPod2", "myPod3"])
Mount	Mount will attach POSIX compliant storage to a pod.	mount.Device(myPod, s3, "/data")

Authors: Kris Nóva

Aurae Language

The Aurae language is a turing complete alternative to YAML that ships with the memory safety and runtime guarantees as the Rust programming language.

```
#!/usr/bin/env aurae

let helloContainer = container();
helloContainer.image("busybox");


let helloPod = pod();
helloPod.env("key", "value");
helloPod.env("foo", "bar");
helloPod.expose(80);
helloPod.expose(8080);

helloPod.add(helloContainer);

let aurae = connect();
let runtime = aurae.runtime();
runtime.run(helloPod);
```

Manifests

Application owners and platform engineers will use the same language to represent static applications that can be used to mutate a system. In other words aurae files can reference each other. Application teams will be able to define their components without taking action with them just by providing a static file with their application needs.

Cluster specific configuration will be managed using the bath subsystem. Application owners can build their applications however they like. The infrastructure specific changes will come later in a build pipeline.

Authors: Kris Nóva

Appendix

The Architecture tries to define and reference resources whenever possible. We do most of that here in the Appendix.

Cluster vs Mesh

We do our best not to refer to a group of Aurae nodes as a **cluster** but rather as a **mesh.** We do this in order to outline the difference in the peer to peer relationship between nodes, and the organic growth paradigm of Aurae.

Decision to stay away from BitTorrent infrastructure

The Aurae project does not use any public bittorrent or libp2p infrastructure in any way.

While the core DHT paradigms might be similar, an Aurae mesh will be responsible for hosting or identifying its own public service discovery infrastructure such as public DNS or a DHT.

Authors: Kris Nóva