ANSM: AGILE NETWORK SUBSTRATE FOR CLOUD-BASED AGENT SANDBOXES

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent sandboxes let AI agents execute code, invoke tools, and access data in isolated, short-lived environments. At cloud-provider scale, networking must be provisioned for hundreds of sandboxes per node within milliseconds and must give each tenant access to its VPC and the Internet. We present ANSM, an agile network substrate that separates a high-density data plane on sandbox nodes from tenant-scoped network services on gateway nodes. Sandbox nodes pool pre-addressed virtual interfaces and use programmable forwarding, so every sandbox sees an identical network inside the guest while retaining a unique identity across the deployment. Gateway nodes hold public IP addresses and VPC interfaces as network services shared by all of a tenant's sandboxes, allowing these resources to be provisioned, scaled, and updated independently of sandbox creation. As a result, sandbox startup does not wait for slow cloud-resource provisioning, while public addresses and VPC interfaces remain isolated across tenants. In production, ANSM serves a rapidly growing sandbox workload, setting up each sandbox's network within milliseconds and supporting hundreds of sandboxes per node.

1 INTRODUCTION

Language-model agents increasingly accomplish tasks by executing code. They run programs, invoke tools, and operate on data in automated loops [1, 2]. Each task runs in an *agent sandbox*, an isolated, ephemeral environment created for the task and discarded when it ends. The cloud is a natural home for this workload: it absorbs bursts by pooling capacity across tenants and already hosts the private repositories, datasets, and internal services that agents need to access. *Networking*, however, exposes a mismatch between this workload and conventional cloud infrastructure. Each sandbox must reach the Internet and its tenant's private network (VPC) without requiring per-instance network configuration inside the sandbox. These requirements pull the design in opposing directions. Speed and density favor identical sandboxes with networking prepared in advance, whereas multi-tenancy and flexible connectivity require each sandbox to be associated with its tenant, reach authorized VPC resources, and access the Internet through tenant-specific addresses.

Specialized vendors such as E2B [3] and Daytona [4] build sandbox services on public cloud infrastructure. Their tenant-agnostic sandbox model simplifies networking, but their limited control over the underlying cloud infrastructure reduces opportunities for end-to-end optimization. A multi-tenant cloud platform must additionally address two challenges. First, VPC access requires explicit tenant binding. A sandbox's local interface alone does not establish access to its tenant's VPC. Without an integrated private-network path, tenants need either to expose private resources through public endpoints or to operate additional customer-managed proxy infrastructure, such as E2B's proxy-tunneling mechanism [5]. Second, a sandbox's public egress identity depends on the node where it runs. When sandboxes share a node-level public IP, tenants also share its reputation, so restrictions on that IP can affect other tenants. Assigning each sandbox a dedicated public IP improves isolation, but places cloud-resource provisioning on the startup path.

We present ANSM, the network substrate behind agent sandboxes deployed in production at a leading cloud service provider. ANSM resolves the tension above by separating reusable sandbox node networking from tenant-specific network resources. Sandbox nodes contain only identical, pre-created interfaces that carry no tenant identity. Tenant-specific resources are instead placed on

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

gateway nodes, where they are provisioned once per tenant and shared across all of that tenant's sandboxes. Our main contributions are as follows:

• We design the *sandbox node* around reusable interfaces, preassigned addresses, and a shared programmable data plane to move most network provisioning off the sandbox startup path. Reusable interfaces with preassigned addresses eliminate per-sandbox interface creation and address allocation. The shared programmable data plane keeps sandbox-specific forwarding state and policies outside the guest, allowing every sandbox to boot from the same network template. Together, these choices allow a node to host hundreds of sandboxes from mutually untrusted tenants, provision each sandbox's network in milliseconds, and avoid per-sandbox growth in host routing and filtering state.

• We introduce tenant-scoped *network services* that provide a unified resource model for VPC and Internet access. Each service's interfaces and addresses reside on *gateway nodes*, while sandbox nodes simply bind each sandbox to its tenant's services. This decouples tenant network resources from individual sandbox lifecycles and from the nodes that host them. Sandboxes on different nodes can share the same tenant services, while those on the same node can reach different VPCs and use different public IP addresses. Because services are provisioned per tenant rather than per sandbox, a small set can serve a large sandbox population, while tenants can add or change their public IP addresses without affecting sandbox creation or network configuration.

2 Background and Related Work

Agent sandbox. Two properties shape how agent sandboxes are built: containment and short lifetime. *i)* Because the code a sandbox runs is unreviewed, the sandbox must keep erroneous or adversarial actions away from the host and other workloads. *ii)* A sandbox runs for seconds to minutes and is discarded once its task ends, so no state carries over to the next one. These requirements make conventional containers and VMs imperfect fits. Containers such as runc [6] start quickly but share the host kernel, while traditional VMs provide stronger isolation at the cost of higher startup overhead. *MicroVMs* combine hardware virtualization through KVM with a minimal device model and guest image, giving each sandbox its own kernel while retaining near-container startup performance. Firecracker, for example, reports about 125 ms startup and a few megabytes of memory overhead per microVM [7]. Kata Containers [8] and Cloud Hypervisor [9] follow similar designs. Commercial platforms such as E2B [3] and Daytona [4] build their execution environments on microVMs.

Network requirements for agent sandboxes. Containment requires network isolation, while short lifetimes make network setup a recurring cost. Together, they impose the following requirements beyond those of conventional cloud networking.

i) Density and Elasticity: A single physical node must host hundreds to thousands of concurrent sandboxes, each with its own virtual interface, address, and forwarding rules. Because sandboxes are short-lived, these resources must be provisioned and reclaimed continuously without adding perceptible latency to sandbox startup.
ii) Multi-Tenancy Network Accessibility: Sandboxes that share a compute node must not be forced to share VPC access or a public IP address. Unlike a persistent web server, whose clients typically require a stable inbound address, an agent sandbox often needs outbound access to package repositories, external APIs, and data inside its tenant's VPC. Since sandboxes execute model-generated code, this access should be enforced outside the sandbox rather than by code running inside it.
iii) Network Homogeneity: Every sandbox should present an identical guest-side network configuration, including its IP address, routing table, and DNS settings. Snapshot-based microVM startup avoids repeating guest and application initialization [10, 11]. An identical network configuration lets sandboxes reuse the same snapshots across hosts without reconfiguring the guest network.

Existing networking solutions. Conventional cloud networking typically gives each workload a distinct network interface, such as an *Elastic Network Interface* (ENI). Because these interfaces are hardware-backed, their attachment latency and per-node limits constrain both provisioning speed and density. Software-defined data planes such as AWS Hyperplane [12] and Google Andromeda [13] improve scalability by decoupling network state from physical devices. However, each workload remains a distinct network endpoint with its own address and forwarding state.

Kubernetes *Container Network Interface* (CNI) plugins such as Calico [14], Cilium [15], and Flannel [16] connect pods through software-defined virtual interfaces. However, network provisioning

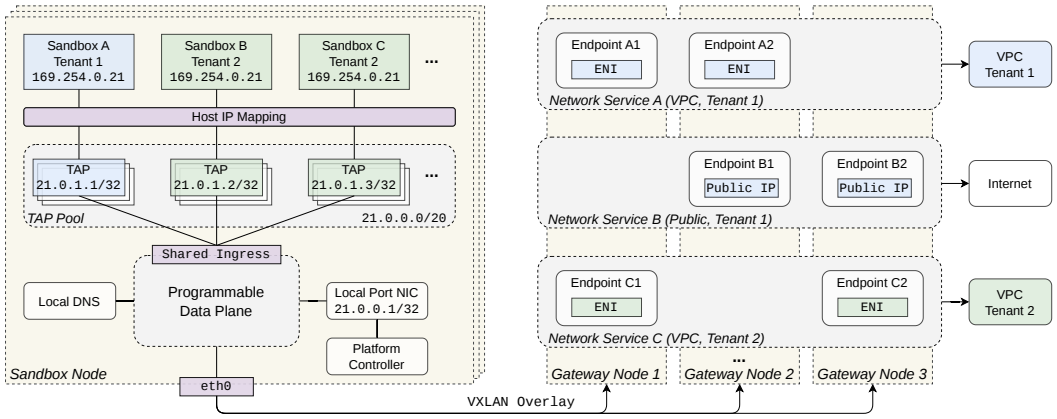

Figure 1: Architecture of ANSM.

remains tied to the pod lifecycle and the cluster control loop, so per-pod state becomes costly to reconcile when sandboxes are created and destroyed at a high rate. Kubernetes also assumes individually addressed pods in a largely flat network [17], so supporting sandbox homogeneity, accessibility, and multi-tenancy requires additional per-sandbox network state and configuration.

Purpose-built sandbox platforms come closest to meeting these requirements by providing *microVM-native interfaces*. AgentENV [18] and E2B [3], for example, provide microVM connectivity with network state managed locally on each host. This avoids cluster-wide coordination and keeps provisioning inexpensive, but still introduces per-sandbox network state and setup overhead. More fundamentally, these systems run on public clouds that they do not operate, which limits their control over the underlying infrastructure and their capability for end-to-end optimization. For instance, to access a tenant's VPC, E2B requires either tenant-operated proxies or a BYOC deployment inside that VPC [19, 20]. In addition, sandboxes from different tenants may share an egress IP, so address-based restrictions triggered by one tenant can affect the others.

3 System Design of ANSM

3.1 Architecture Overview

Fig. 1 shows the architecture of ANSM, which consists of two types of nodes. *Sandbox nodes* are bare-metal machines that run sandboxes and provide reusable virtual interfaces and a shared programmable data plane. *Gateway nodes* host tenant VPC interfaces and public IP addresses, through which sandboxes reach networks outside the platform. This division allows tenant network resources to be provisioned in advance and shared across sandboxes, keeping their allocation off the sandbox creation path. It also removes the need to allocate an ENI or public IP for every sandbox, easing quota constraints on sandbox density.

Each sandbox node is delegated an address block in advance (*21.0.0.0/20* in the figure) and prepares a pool of kernel virtual interfaces (Linux TAP devices [21]). Because the blocks are disjoint, addresses drawn from them are unique across the deployment without coordination at allocation time. Each interface carries a preassigned platform address and is bound to a sandbox on demand. Every sandbox nonetheless observes the same guest-visible address (*169.254.0.21* in the figure). Host-side IP mapping translates between the guest and platform addresses, so each sandbox remains distinguishable outside the guest. All pooled interfaces connect to a programmable data plane, which holds sandbox-specific forwarding rules and directs traffic to gateway nodes through overlay tunnels such as VXLAN [22]. For management, a sandbox controller reaches sandboxes through a separate host-side port.

External reachability is provided by *network services* rather than by interfaces attached to individual sandboxes. A network service represents access by one tenant to a VPC or the Internet and is backed by a set of interchangeable endpoints. Each endpoint holds either an ENI into the target

VPC or a public IP for Internet egress. Endpoints are grouped into services independently of where they run: replicas of one service can run on different gateway nodes, while one gateway node can host endpoints from different services. This allows services to scale flexibly and supports recovery from gateway-node failures. Fig. 1 shows a public network service for tenant 1 with two public-IP endpoints, and a VPC network service for each of tenants 1 and 2, each with two endpoints holding ENIs into that tenant's VPC. A tenant's sandboxes share the endpoints of its services across sandbox nodes, while no endpoint is shared across tenants.

3.2 Sandbox Node: Fast Provisioning and High Density

A sandbox node must host hundreds of sandboxes while provisioning each sandbox's network within milliseconds. Neither goal is achievable if device creation, address assignment, and route installation occur per sandbox, because these operations lie on the startup critical path and their state grows with the number of sandboxes. We address this challenge through the following design choices.

Namespace-free network provisioning. Container-oriented designs place each sandbox in a separate network namespace (*netns*). This adds the overhead of creating a namespace and veth pair for every sandbox and scatters policy across per-sandbox rule sets. We instead use a single programmable data plane per node, reducing per-sandbox setup to binding a pre-created interface and installing its flow rules. Each sandbox therefore requires only one virtual interface with a unique address in the host netns.

Interface pooling. We make two choices for each sandbox's network interface. First, we use TAP devices instead of ENIs. TAP devices are kernel objects, so a node can maintain thousands of them without consuming its limited hardware-backed ENI quota. Second, we create these devices in advance. Although creating a single TAP device is inexpensive, short sandbox lifetimes make repeated creation costly. Each node therefore creates a pool of 4,000 TAP devices at initialization. Sandbox creation binds an interface from the pool, and termination returns it.

Address preassignment. Like interfaces, addresses are assigned in advance. The control plane assigns each node a non-overlapping */20* block of 4,096 IP addresses. When a node builds its TAP pool, it binds one address from its block to each device. Every sandbox therefore receives a platform-unique address, with no coordination needed at startup. Gateway and control components can identify a sandbox by this address alone, without a separate node identifier. In addition, each sandbox uses a */32* prefix, allowing the programmable data plane to forward traffic directly to its interface without a shared subnet or link-layer ARP resolution.

Guest homogeneity. In snapshot-based microVM startup, every sandbox boots from the same network template, with the same address and routes. All differences between sandboxes are therefore handled outside the guest. On each TAP device, stateless IP rewriting maps between the shared guest-visible address and the device's platform-unique address. The programmable data plane then uses this unique address to enforce the sandbox's reachability and access policies. As a result, no instance-specific network configuration needs to be written into the guest at startup. From a networking perspective, guest homogeneity also makes sandboxes forkable and migratable.

Asynchronous provisioning. Since no configuration needs to be written into the guest, network setup can be split into two steps. The first step, device binding, attaches a device from the pool to the sandbox. It is the only step the hypervisor requires before booting the guest, and it takes about one millisecond. The second step, state installation, programs the sandbox's forwarding and policy rules. This step is more expensive, but it changes only flow rules outside the guest and is not needed until the sandbox sends its first packet. We therefore run it concurrently with microVM initialization, so that only device binding remains on the startup critical path.

3.3 Gateway Node: Scalable and Isolated External Access

Unlike the TAP devices in Sec. 3.2, public IPs and ENIs are slow to provision, limited in number, and tied to a tenant. They are provisioned through the cloud control plane, which can take tens of seconds per attachment [23]. ENIs are hardware-backed and limited per machine, typically to a few dozen interfaces or a few hundred with trunking [24]. These resources also carry tenant identity, so they cannot be shared freely across tenants. Attaching them directly to sandbox nodes would

therefore put a slow, tenant-specific operation on the sandbox startup critical path and cap each node's sandbox population at its ENI limit.

Relocating network resources. We therefore move public IPs and ENIs off sandbox nodes and onto gateway nodes, where they are managed as *network services* (Sec. 3.1). A tenant's network services are created when the tenant subscribes to the sandbox service and binds network resources to its account, rather than when each sandbox starts. The resources themselves are held by the service's *endpoint* replicas, which are provisioned once, ahead of sandbox creation. Sandboxes of the same tenant can then share these endpoints, allowing a few ENIs per tenant to serve thousands of sandboxes. This design eliminates the need to attach a cloud interface to each sandbox or sandbox node and decouples cloud network-resource provisioning from sandbox lifecycles.

Transparent address rotation and scaling. Agent workloads can generate bursts of outbound requests that trigger remote rate limits or blocks on source IPs. A public network service therefore exposes a pool of public IP addresses through its endpoints. Addresses can be withdrawn, replaced, or added dynamically without interrupting running sandboxes. Neither endpoint rotation nor replica scaling requires any configuration change inside the sandbox, because its guest-visible address stays fixed (Sec. 3.2) regardless of which public address carries its traffic.

Access to tenant private networks. Sandboxes often need to access code repositories, datasets, and logs that reside in the tenant's VPC. The same network service abstraction supports this access. A VPC network service is backed by endpoints that hold ENIs attached under a role granted by the tenant. As with public services, these endpoint replicas can scale with throughput demand independently of the sandboxes that use them. By contrast, platforms such as E2B require either deployment in the tenant's own cloud through BYOC [20] or a tenant-operated proxy tunnel [19] for VPC access. Otherwise, private resources must be exposed through public endpoints.

Structural tenant isolation. Because network services are scoped to tenants, isolation follows from the design rather than from configuration. No two tenants share a public network service or public IP address, so an IP-based restriction triggered by one tenant does not affect other tenants. In contrast, a shared or node-scoped address pool offers no such guarantee. Private access is isolated similarly, since a sandbox can reach a VPC only through a network service associated with its tenant.

4 Implementation of ANSM

This section describes how we implement the design in Sec. 3. Sec. 4.1 covers the forwarding path on a sandbox node, from flow-rule lookup in the programmable data plane to the VXLAN tunnels that carry traffic to network services. Sec. 4.2 presents optimizations that keep per-node state and load small as the sandbox population grows. Finally, Sec. 4.3 describes the Kubernetes-based network service management, failure recovery, image pulls, and inbound access.

4.1 Implementation of Sandbox Nodes

Source-keyed flow rules. We implement the programmable data plane entirely as OpenFlow [25] match-action rules in the node's Open vSwitch (OVS) [26] instance. Each rule matches two fields: *i)* the source address of the TAP interface, which is unique across the platform (Sec. 3.2) and therefore identifies the sandbox and its tenant, and *ii)* the destination prefix, which classifies the target as public or private. The rule's action forwards the packet to the corresponding network service. Because the source address uniquely identifies the sender, no tag or per-packet state is needed to establish packet origin.

Connection type. Tenants do not program these rules directly. Instead, each sandbox declares a *connection type* at creation, which specifies the network services it may use. The control plane then installs the corresponding rules during asynchronous provisioning (Sec. 3.2). An `INTERNET_VPC` type networking sends public destinations to its tenant's public network service and private prefixes [27] to its VPC network service. `INTERNET` and `VPC` allow only the corresponding destination class, while `NONE` blocks both. For sandboxes that need finer control, such as access to multiple VPCs, `ROUTE_DEFINED` accepts an explicit route table and compiles each entry into a forwarding rule. For example, one entry may send *10.0.0.0/8* to network service A and another *172.16.0.0/12* to network service B.

Rule aggregation. Each sandbox requires only a handful of flow rules. A fixed connection type uses a few rules, including those for the return path and drop policy, while `ROUTE_DEFINED` adds one rule per route-table entry. Even a fully loaded node therefore holds only on the order of ten thousand rules, well within the capacity of a single OVS instance. Because the rules match coarse destination classes rather than individual connections, OVS can cache each forwarding decision as a wildcarded entry. Cached entries then serve every connection from a sandbox to the same destination class, rather than requiring one exact-match entry per connection. The rules remain fixed throughout the sandbox's lifetime, installed at creation and removed at termination.

Overlay to network services. Sandbox addresses are internal to the deployment and are not routable across the physical network. The data plane therefore encapsulates packets destined for network services in VXLAN [22], using the sandbox node's underlay address as the outer source and the selected gateway node's underlay address as the outer destination while leaving the inner packet unchanged. The gateway node decapsulates the packet, and the endpoint applies source NAT, translating the sandbox's platform address to the endpoint's public IP or ENI address. The destination network thus sees a routable address for replies. The Linux kernel's connection-tracking and NAT mechanisms maintain this mapping for subsequent packets and reverse the translation for replies [28]. The gateway then tunnels the replies back to the sandbox node. Sandbox nodes track endpoint health and select a healthy endpoint for each new connection, allowing endpoints to change without reconfiguring the guest. This design is not specific to OVS. The same *match-action* and overlay logic can be implemented in eBPF [29], potentially reducing per-packet overhead on recent kernels.

4.2 Optimization of Sandbox Nodes

Port aggregation. If each of the 4,000 pooled TAP devices had its own OVS port, the switch would have to manage thousands of ports. Instead, a thin forwarding layer directs traffic from all TAP devices to a single shared OVS ingress (Sec. 3.1). This does not affect sandbox identification, since flow rules match the source address rather than the ingress port. As a result, the number of OVS ports remains constant regardless of the pool size or the number of active sandboxes.

ARP elimination. Placing thousands of sandboxes on a shared subnet would require the host to maintain a neighbor entry for each sandbox. Instead, both ends of each sandbox link use `/32` prefixes (Sec. 3.2), and ARP is disabled in the sandbox network template. The node's data plane forwards packets using explicit interface and IP mappings rather than dynamic neighbor resolution. Together, these choices keep both neighbor state and ARP broadcast traffic off the node.

Local DNS caching. With hundreds to thousands of sandboxes per node, DNS resolution alone can exceed the provider's resolver quota of roughly 5,000 queries per second per node [30]. Sandboxes of the same tenant often access the same destinations, making their DNS queries highly cacheable. The same pattern holds for package installation during startup, when many sandboxes fetch the same `pip` and `apt` packages. We therefore deploy a node-local DNS cache to serve repeated queries, reducing both upstream DNS traffic and resolution latency.

4.3 Implementation of Gateway Nodes

Network services as Kubernetes resources. Each network service is represented by a custom resource backed by a Kubernetes *Deployment*, whose endpoint *Pods* are scheduled onto gateway nodes. Each endpoint Pod has two interfaces: one on the platform network, which terminates VXLAN tunnels from sandbox nodes, and one carrying the tenant's ENI or public IP, which forwards traffic after decapsulation and source translation (Sec. 4.1). This mapping greatly simplifies implementation by reusing Kubernetes's mature scheduling, autoscaling, and recovery mechanisms for reliable network service management. Each endpoint contributes one address, so scaling a network service amounts to adjusting its Deployment's replica count. Traffic metrics can drive this adjustment through a Horizontal Pod Autoscaler (HPA), while topology spread constraints distribute replicas across gateway nodes. Pod health probes report endpoint readiness and liveness, and Kubernetes restarts failed containers and replaces failed Pods to maintain the desired replica count.

Address-preserving recovery. Endpoint failure is handled without operator intervention or reconfiguration. When an endpoint Pod fails, Kubernetes recreates it. Because the ENI or public IP persists independently of the Pod, the replacement reattaches the same resource and retains the same

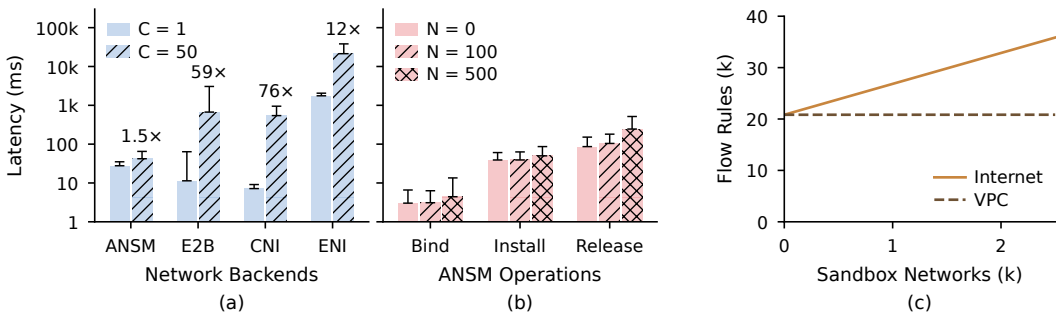

Figure 2: Evaluating the density and elasticity of ANSM. **(a)** Network setup latency of each backend under C concurrent requests, with the slowdown from $C{=}1$ to $C{=}50$ marked above each pair. **(b)** Latency of ANSM operations under 50 concurrent requests with N sandboxes already running on the node. **(c)** Flow rules on the node's virtual switch as sandbox networks are provisioned. In (a) and (b), each bar shows the mean latency, with the whisker marking the 99th-percentile latency.

address. Remote peers and allow lists therefore require no updates. Connections in flight through the failed endpoint are lost because their connection-tracking state resides in that Pod, but retried new connections continue to use the same set of addresses. Meanwhile, the sandbox node steers new connections away from the failed endpoint until its replacement becomes healthy (Sec. 4.1).

Cross-VPC registry access. Tenants can customize sandbox images and store them in registries inside their own VPCs. Without a path into the VPC, tenants may need to copy these images to platform-accessible storage in advance. With ANSM, the tenant's VPC network service can be prepared before sandbox creation, allowing the platform to pull custom images directly from the registry during startup. Image retrieval and runtime VPC access therefore reuse the same mechanism.

Inbound access. The platform controller reaches a sandbox directly at its platform-unique address through the node local port (Sec. 3.1). External access uses a different path. Because sandboxes are short-lived and rarely host persistent services, they receive no dedicated public IP addresses for inbound access. Instead, each sandbox is assigned a hostname derived from its identifier, and an Envoy [31] reverse proxy forwards HTTP and WebSocket requests for that hostname to the sandbox.

5 Evaluation

Experimental setup. We evaluate ANSM's scalability, network accessibility, and data-plane overhead both in controlled environments and in production. Controlled experiments are conducted on a commodity bare-metal server with 2x Intel Xeon Platinum 8269CY processors (104 vCPUs) and 192 GiB of memory. Each sandbox runs as a microVM booted from the same base image. Production measurements are collected from two public cloud regions where ANSM supports a commercial agent-sandbox service.

Density and elasticity. We examine whether ANSM can provision networks quickly when many sandboxes start concurrently and when a node already hosts many sandboxes. We compare ANSM with three widely used alternatives. E2B's network pool [3] and the standard CNI bridge plugin [17] run on the same lab server, while a per-sandbox cloud ENI is evaluated on a bare-metal cloud instance. For ANSM, setup consists of device binding followed by forwarding and policy-state installation. E2B allocates a network from its pool, CNI creates a namespace and invokes the plugin's ADD operation, and ENI issues cloud API calls to create and attach an interface. Fig. 2(a) shows that, with one request at a time, all backends except ENI provision a network within tens of milliseconds. Under 50 concurrent requests, however, E2B's mean setup latency increases 59× to 669 ms, with a P99 of 3.1 s. By default, E2B maintains 32 pre-created networks in addition to networks recycled from exited sandboxes. When no recycled network is immediately available and the pre-created pool is exhausted, a request must wait for a new network to be created, which can take seconds. To determine whether additional pooling alleviates this bottleneck, we increase E2B's pool size eightfold. This reduces the mean latency under 50 concurrent requests by 62%, but the P99 remains about 3.3 s because some requests still require on-demand network creation. The larger pools also increase

Table 1: Network accessibility of sandbox platforms. ● denotes a native capability; ◑ denotes a capability available only with additional setup, such as an enterprise deployment, customer-operated proxy, or infrastructure-specific network configuration; and ○ indicates an undocumented capability.

	Capability	AgentENV	E2B	K8s Pod	**ANSM**
Private	Tenant VPC Connectivity	◑	◑	◑	●
	Multi-VPC Connectivity	○	◑	◑	●
Public	Tenant-dedicated Egress IP Pool	○	◑	◑	●
	Runtime Egress IP Switching	○	◑	◑	●

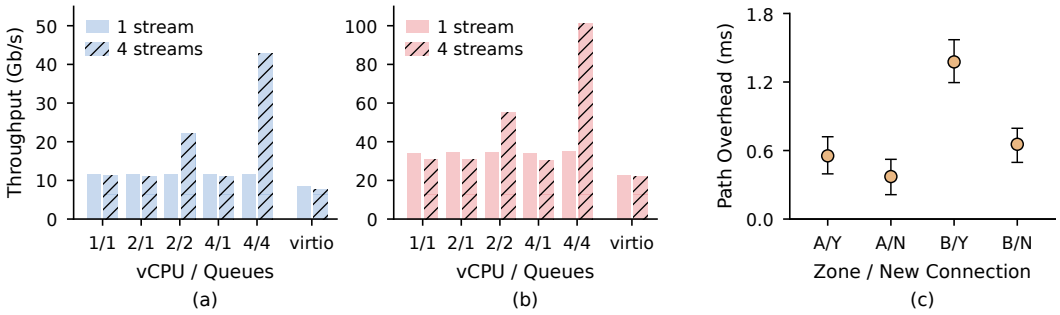

Figure 3: Evaluating the overheads of ANSM. TCP throughput **(a)** from *sandbox node* into sandbox and **(b)** from sandbox to *sandbox node* under different vCPU, queue, and stream configurations. **(c)** Extra path overhead incurred by ANSM when sandboxes in zone A (same-zone) and zone B (cross-zone) reach a fixed endpoint in the *gateway node*, over new (Y) and reused (N) TCP connections.

warm-up time from 2.7 to 17 s. In contrast, ANSM's mean setup latency increases only $1.5\times$, from 28 to 42 ms, because device binding draws from a pre-created pool and state installation requires only adding flow rules. State installation also overlaps with microVM initialization, keeping most of its latency off the startup critical path (Sec. 3.2). Fig. 2(b) shows that ANSM remains fast as the node fills. With 500 sandboxes already running, device binding and state installation take only 4.4 and 50 ms on average, respectively, under 50 concurrent requests. Network release latency increases from 86 to 245 ms, but release occurs after the sandbox exits and therefore does not delay startup. Fig. 2(c) shows that forwarding state also grows modestly with sandbox count. The node's 2,500 pre-created devices contribute a fixed base of 20,827 flow rules. Each `INTERNET` sandbox adds six rules, bringing the total to 35,827 at 2,500 networks, whereas `VPC` sandbox networks add almost no per-sandbox rules because they share tenant-level network-service rules. Allocation latency remains nearly flat as the number of networks increases. In practice, sandbox density is limited primarily by CPU and memory rather than networking. A pool of 4,000 devices sustains allocation and reclamation without becoming a bottleneck.

Network accessibility. Table 1 compares ANSM with AgentENV [18], E2B [3], and an ordinary Kubernetes Pod [17] based on their documented user-facing capabilities. AgentENV can reach private address ranges through deployment-specific routing and provides Internet access through host NAT, but it documents neither a VPC-attachment abstraction nor tenant-specific egress-IP management. E2B supports private VPC access through Enterprise BYOC and can use customer-operated proxies for private connectivity and customer-owned egress addresses, but these capabilities are not native to its standard hosted service. An ordinary Kubernetes Pod similarly depends on the surrounding CNI, cloud routing, and egress configuration for VPC access and public-address management. In contrast, ANSM provides all four capabilities natively through tenant-scoped network services. A sandbox can reach its tenant's VPC without a guest-side proxy or tunnel, and prefix routes can steer different destinations to multiple VPC services. A tenant can also use a dedicated public network service backed by multiple egress addresses. Endpoints can be added or withdrawn without restarting sandboxes, so new connections can move to different addresses while established connections remain pinned to their existing endpoints.

Performance trade-offs. The benefits above come with additional data-plane overhead. ANSM places a TAP device and a programmable forwarding path between each sandbox and the physi-

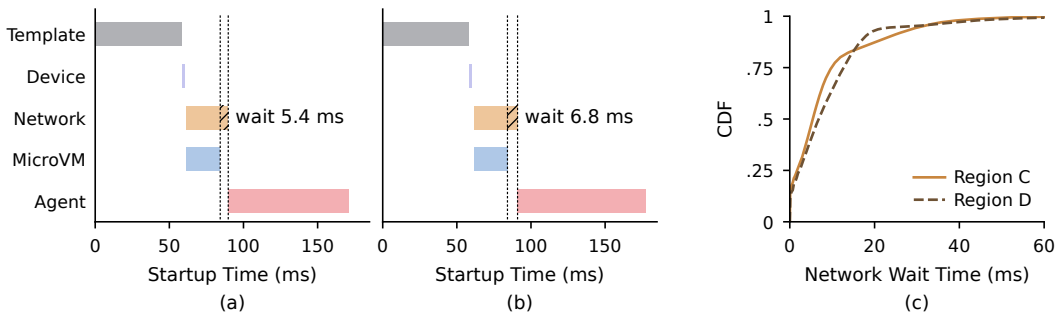

Figure 4: Representative startups from **(a)** Region C and **(b)** Region D, with network wait times close to the median in each region. Each startup loads the sandbox template, binds a device, installs forwarding rules (Network) concurrently with microVM initialization, and then waits for the guest agent. The hatched segment denotes the network wait time, during which network-state installation extends beyond microVM initialization. **(c)** CDF of network wait time across all recorded startups.

cal network, while external traffic traverses a shared gateway endpoint rather than a per-sandbox interface. Fig. 3(a,b) evaluates the throughput of this path for a single sandbox and identifies the effect of parallelism. Throughput increases substantially only when the number of queues grows with the number of streams: four streams on a single queue provide little or no improvement over one stream, whereas four streams over four queues reach 42.8 and 101.0 Gb/s into and out of the sandbox, respectively. This is because each queue is backed by a separate vhost worker on the host and a separate interrupt vector in the guest, allowing concurrent flows to be processed in parallel rather than funneled through a single worker. Even with a single queue, ANSM sustains 11.5 and 34 Gb/s, far exceeding the fraction of a node's physical-link capacity that an individual sandbox can draw on when thousands of sandboxes share the node. Fig. 3(c) quantifies the latency cost of routing external traffic through a gateway node rather than directly through the sandbox node's interface. With 95% confidence, the additional path latency is 0.21–0.72 ms when the gateway is in the same cloud zone and 0.50–1.57 ms when gateway nodes are distributed across zones for availability. New connections incur greater overhead than reused connections because connection establishment must also traverse the gateway path. Overall, ANSM trades a low-millisecond increase in path latency for the provisioning, isolation, and resource-sharing benefits described above.

Provisioning latency in production. Finally, we examine whether network provisioning delays sandbox startup in production, using over $330{,}000$ startup records collected in two regions of ANSM's deployment. Fig. 4(a,b) shows a representative startup from each region. After loading the sandbox template, the node binds a device in about 2 ms and then installs the sandbox's forwarding and policy state concurrently with microVM initialization (Sec. 3.2). The guest agent starts only after both operations complete. When microVM initialization finishes first, startup must wait for network installation, and we refer to this residual delay as the *network wait time*. MicroVM initialization in our production system has improved to about 20 ms, which is now shorter than the roughly 30 ms needed for network installation. As a result, the network wait time averages 8.5 and 9.3 ms in Regions C and D, accounting for 3.7% and 5.0% of total startup time, respectively. The corresponding P99 values are 46.9 and 54.5 ms (Fig. 4(c)). To fully eliminate this residual delay from the critical path, we are developing a strategy that begins network installation while the sandbox template is still being loaded.

6 Conclusion

In this paper, we presented ANSM, a comprehensive network solution designed for short-lived agent sandboxes at cloud scale. By separating reusable sandbox-node networking from tenant-specific network services, ANSM provides millisecond-scale network provisioning, high sandbox density, isolated VPC and Internet access, and a uniform guest-side network configuration, all with low overhead. ANSM is deployed in production at a major cloud provider, where it supports a large volume of sandbox startups per day while sustaining hundreds of sandboxes per node.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, assist with translation, and interpret results. We did not use generative AI tools to propose or refine hypotheses, or to design or provide feedback on research methodology or experiments. The remaining tasks requiring disclosure are not applicable to this work: generating synthetic datasets, developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs, cleaning and reformatting datasets, and supporting qualitative and thematic data analysis. Additionally, we used generative AI tools to create or edit software code, draft parts of this paper, edit it to improve readability, create or modify figures, transcribe recordings of research meetings, summarize existing literature, and identify relevant literature. We have reviewed all AI-assisted work: all LLM-drafted text was rewritten and proofread by the authors, all LLM-generated code was verified and tested, and all factual claims about related systems were checked against their primary documentation. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of AI.

REPRODUCIBILITY STATEMENT

ANSM is deployed in production at a commercial cloud provider, where it provides networking for a publicly available agent-sandbox product. Users of this product already rely on the mechanisms described in this paper, including millisecond-scale sandbox startup, identical guest-visible network configuration across sandboxes, and direct access to tenant VPCs. We omit the provider's name for double-blind review and will disclose it in the camera-ready version.

The experiment section reports the hardware, sandbox specifications, concurrency levels, background sandbox populations, and production measurements from live traffic. ANSM provisions ENIs and public IP addresses through the cloud provider's public APIs. Reproducing the full system requires access to the corresponding cloud resources. The design section and the implementation section provide sufficient details to reproduce the sandbox-node data plane using standard components, including TAP devices, Open vSwitch, VXLAN, and Kubernetes.

REFERENCES

[1] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji, "Executable code actions elicit better LLM agents," in *Proceedings of the 41st International Conference on Machine Learning*, vol. 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232, PMLR, 2024.

[2] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems*, vol. 37, 2024.

[3] E2B, "Internet access - documentation." `https://docs.e2b.dev/network/internet-access#sandbox-public-url`. Accessed: Sep. 2026.

[4] Daytona, "Daytona - secure infrastructure for running ai-generated code." `https://www.daytona.io/docs/sandboxes`. Accessed: Sep. 2026.

[5] E2B, "Proxy tunneling." `https://docs.e2b.dev/network/ip-tunneling`. Accessed: Sep. 2026.

[6] Open Container Initiative, "runc: Cli tool for spawning and running containers." `https://github.com/opencontainers/runc`. Accessed: Sep. 2026.

[7] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *NSDI 2020*.

[8] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling mec services in fast and secure way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pp. 209–214, IEEE, 2019.

[9] Cloud Hypervisor, "Cloud hypervisor: A virtual machine monitor for modern cloud workloads." https://github.com/cloud-hypervisor/cloud-hypervisor. Accessed: Sep. 2026.

[10] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 559–572, Association for Computing Machinery, 2021.

[11] N. Lazarev, V. Gohil, J. Tsai, A. Anderson, B. Chitlur, Z. Zhang, and C. Delimitrou, "Sabre: Hardware-accelerated snapshot compression for serverless MicroVMs," in *18th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '24, pp. 1–18, USENIX Association, 2024.

[12] C. Munns, "Announcing improved vpc networking for aws lambda functions." https://aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions/. Accessed: Sep. 2026.

[13] M. Dalton, D. Schultz, A. Arefin, *et al.*, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *NSDI 2018*.

[14] Tigera, "Project calico: Networking and network security for kubernetes." https://www.tigera.io/project-calico/. Accessed: Sep. 2026.

[15] Cilium Authors, "Bpf and xdp reference guide." https://docs.cilium.io/en/stable/reference-guides/bpf/. Accessed: Sep. 2026.

[16] Flannel-io, "Flannel: A network fabric for kubernetes." https://github.com/flannel-io/flannel. Accessed: Sep. 2026.

[17] The Kubernetes Authors, "Services, load balancing, and networking." https://kubernetes.io/docs/concepts/services-networking/. Accessed: Sep. 2026.

[18] kvcache-ai, "AgentENV: A distributed platform for running agent environments at scale." https://github.com/kvcache-ai/AgentENV. Documentation: https://kvcache-ai.github.io/AgentENV/. Accessed: Sep. 2026.

[19] E2B, "Bring your own proxy (BYOP)." https://docs.e2b.dev/network/byop. Accessed: Sep. 2026.

[20] E2B, "Bring your own cloud (BYOC)." https://docs.e2b.dev/byoc. Accessed: Sep. 2026.

[21] The Linux Kernel Development Community, "Universal TUN/TAP device driver." https://docs.kernel.org/networking/tuntap.html. Accessed: Sep. 2026.

[22] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks," RFC 7348, RFC Editor, Aug. 2014.

[23] Xiheng, "How does alibaba cloud build high-performance cloud-native pod networks in production environments?." https://www.alibabacloud.com/blog/how-does-alibaba-cloud-build-high-performance-cloud-native-pod-networks-in-production-environments_596590. Alibaba Cloud Blog. Accessed: 2026-09-26.

[24] Amazon Web Services, "Supported instances for increased Amazon ECS container network interfaces." https://docs.aws.amazon.com/AmazonECS/latest/developerguide/eni-trunking-supported-instance-types.html. Accessed: Sep. 2026.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[26] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 117–130, USENIX Association, 2015.

[27] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, "Address allocation for private internets," RFC 1918, RFC Editor, Feb. 1996.

[28] Netfilter Project, "Linux netfilter hacking HOWTO: Information for programmers." `https://www.iptables.org/documentation/HOWTO/netfilter-hacking-HOWTO-4.html`. Accessed: 2026-09-26.

[29] The Linux Kernel Development Community, "BPF documentation." `https://docs.kernel.org/bpf/index.html`. Accessed: Sep. 2026.

[30] Alibaba Cloud, "Alibaba Cloud DNS: Limits." `https://www.alibabacloud.com/help/en/dns/pvtz-restrictions-on-use`. Accessed: Sep. 2026.

[31] Envoy Project, "What is Envoy." `https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy`. Accessed: Sep. 2026.