Bee Private Compute

Steve Korshakov
Founding Engineer, Amazon Bee
Amazon

Karan Lyons
Principal Security Engineer
Work performed while at Amazon

Ethan Sutin
Head of AI, Amazon Bee
Amazon

Jonathan Pugh
Sr. SDE AI, Tools & Automation
Amazon

Aleksei Zinchenko
Member of Technical Staff
Amazon

Alessandro Eppacher
Senior Software Development Engineer
Amazon

Billy Jang
Software Development Engineer II
Amazon

Brian Taylor
Software Engineer
Amazon

Nicholas Anderson
Software Development Engineer II
Amazon

Cole Collins
Software Development Engineer
Amazon

Diksha Aurobindo
Software Development Engineer
Amazon

Ali Mirza
Member of Technical Staff
Amazon

Correspondence: `privacy@bee.computer`

Abstract

Bee provides a personal AI assistant that uses frontier AI models over user-controlled real-world context. Privacy is a core system requirement: the architecture is designed to prevent Bee and other Amazon operators, non-attested internal services, and supporting infrastructure from accessing plaintext user data outside the user's cryptographically authorized processing path.

The Bee Private Compute architecture turns that requirement into a verifiable processing model. User devices generate root secrets and release only scoped server-processing keys after verifying the receiving runtime. Data is encrypted before leaving the device and decrypted only within attested Confidential VMs whose software identity users can verify. Unlike a stateless request processor, Bee is designed for a background personal agent that continuously builds context and reasons over time. The system therefore makes server-side state explicit and bounded with short-lived keys, attestation, transparency, service-identity-scoped networking, and cryptographic consent.

This paper reflects Bee's privacy architecture as of the publication date and may be updated to document subsequent improvements to privacy, security, transparency, and resilience.

1 Introduction to Bee Private Compute

Recent frontier models have crossed a threshold enabling meaningfully useful personal agentic AI. To realize the full benefit of personal AI, agents must understand a user across many domains of their life. Bee has created a small wearable device designed to capture real-world context so that the assistant can operate with the continuity and perspective expected of a personal agent. That context is sensitive: it may include conversations about health decisions, financial planning, family matters, and everyday commitments. Bee's privacy architecture starts from a zero-operator-access requirement. No Bee or Amazon operator can read user plaintext by virtue of running the system. The user's device enforces this, not policy. It releases keys only to runtimes it has attested, and every approved runtime's measurements are published to a public transparency log. On-device computation, where all processing occurs within devices in the user's physical possession that can be protected with passcodes, biometrics, or other means of the user's choosing, is the simplest privacy model. Frontier-model quality, however, requires far more compute than any personal device can practically provide. An architecture that extends privacy protections and trust to server-side processing

is therefore needed. Bee's design goal is to make that server-side processing constrained, attestable, and auditable.

Bee addresses this with confidential computing built on the Amazon Web Services (AWS) Nitro System, the hardware and hypervisor platform underlying EC2 instances. In this paper, "Confidential VM" means a Nitro-based EC2 instance running Bee's measured AMI with NitroTPM attestation, following the isolated compute environment pattern AWS documents for EC2 instance attestation[1][2]; the design relies on Nitro's isolation and no-operator-access model. Nitro provides isolated compute, a minimized attack surface, and a published no-operator-access model for EC2 Nitro hosts, EC2 instance memory, and customer data on local encrypted instance storage or encrypted EBS volumes[3][4][5]. NitroTPM adds measured boot and remote attestation for EC2 instances[6], allowing Bee clients to verify the runtime identity of a Confidential VM before sharing server-processing keys. On this foundation, Bee processes user data only within Nitro-isolated Confidential VMs after the client verifies the VM's software identity. User data is encrypted before leaving the device and decrypted only inside these VMs, whose measured configuration is cryptographically bound to the user's TLS connection before any server-processing keys are shared. Bee's security model combines Nitro's confidential-computing properties with Bee-controlled encryption, attestation verification, transparency records, isolated networking, short-lived keys, and consent enforcement.

The architecture is designed so that Bee and other Amazon employees, operators, and non-attested internal systems cannot access plaintext user data through administrative infrastructure paths. Root account secrets are generated by client devices and never persist on Amazon systems. Scoped server-processing keys are released only to approved Confidential VMs, where they are held in VM memory for up to 7 days to facilitate continuous personal AI processing. Confidential VMs run without SSH, SSM, or shell access. There is no service-wide master key. Each user's content is protected by keys derived from a root secret that only that user's devices hold. Enabling remote shell access or changing processing behavior would require replacing the VM image or manifest, which changes its cryptographic attestation and causes conforming user devices to reject the connection. If a user's device has not been active within the key TTL, keys expire from the VM fleet and the backend cannot decrypt user content until the user reconnects and re-shares the relevant keys.

Each component trusts only what it can cryptographically verify. User devices verify VM attestation through cryptographic proofs rooted in the Nitro System before sharing any encryption keys. VMs verify integrity hashes of the code they run. Both verify transparency log records to ensure only approved software configurations process user data. These properties are auditable by parties with access to the source code, manifests, artifact archives, attestation documents, and transparency log records.

1.1 Security Posture at a Glance

Bee's primary privacy properties come from combining client-held secrets, measured infrastructure, and narrow operational access:

- **Encrypted canonical storage**: User content managed by Bee's encrypted storage path is encrypted before persistence; databases and non-confidential caches store encrypted payloads and metadata. Derived search indexes are treated as active-processing caches and are removed when the user's server-processing key is definitively unavailable.
- **No Amazon-held account root secret**: Account root material is generated and recovered through client-controlled secrets. Amazon stores only client-encrypted account-secret enrollments.
- **Attestation-gated key release**: Clients share scoped server-processing keys only after verifying the receiving Confidential VM's certificate, NitroTPM attestation, and transparency proof.
- **Measured runtime identity**: Base AMIs, manifests, containers, model files, downloaded artifacts, bridges, certificates, pinned secret versions, and selected runtime agents are reflected in the VM's measured identity.
- **Short-lived server-processing window**: Server-processing keys are held in Confidential VM memory and Confidential Redis keystores for at most 7 days of inactivity, enabling background intelligence without creating a permanent server-side decryption capability.

- **No interactive operator path**: Confidential VMs are deployed without SSH, SSM, EC2 Instance Connect, or conventional remote shell administration.
- **Consent-bound external release**: External-service sharing is authorized by signed consent records derived from the user's Account Secret and can be revoked to stop future releases.

1.2 Threat Model and Scope

In this paper, "Bee" means the Bee product and the Amazon team that builds and operates it; "Amazon" means Amazon.com, Inc., including Bee; and "AWS" means Amazon Web Services, which operates the Nitro System and the other infrastructure Bee runs on. "Amazon operators" therefore includes Bee engineers, other Amazon employees, and AWS infrastructure operators. Bee's controls address the first two; the third is a property of the Nitro System that Bee relies on.

Bee Private Compute is designed to protect user plaintext from Amazon operators, non-attested infrastructure, accidental exposure in persistent stores, and data release to external services without cryptographic consent (a signed, expiring consent record bound to the user's keys and verified inside the Confidential VM at the point of release; see Section 8). It assumes that the user client performs attestation verification correctly, that the approved code measured into the Confidential VM is code Bee has qualified for release, with its artifacts permanently recorded in the transparency log, and that derived server-processing keys are shared only after verification succeeds.

Bee's privacy architecture is built on Nitro's isolation and attestation model, then adds Bee-specific controls for the layers that determine which software can receive user keys and how plaintext is handled after release. The security analysis covers Bee-controlled layers: base images, manifests, measured artifacts, attested TLS, key release, application code, key retention, network routing, logging, diagnostics, external integrations, cryptographic consent, and operational access.

The core guarantee is intentionally scoped: plaintext processing is limited to approved software running in attested Confidential VMs after the client releases scoped keys. Approved application code necessarily sees plaintext while performing the service. Bugs in that approved code, compromised client devices, compromised mobile operating systems, unauthorized approval of malicious code, and data retained by external services after an authorized release require defense in depth beyond encryption alone. Bee addresses those risks with measured deployments, transparency records, network isolation, operational controls, audits, consent expiry, and short server-processing key lifetimes.

Threat	Primary Risk	Bee Control
Amazon operator access	Operator reads user plaintext through administrative access	No SSH, SSM, or shell path on Confidential VMs; client releases keys only after attestation; non-confidential stores hold encrypted content; encryption and decryption occur only on the user's device or within attested Confidential VMs
Non-attested service access	Backend, database, cache, or integration code outside the trusted path receives plaintext	Content encrypted before persistence; server-processing keys are scoped to attested VMs; internal Redis/search access is constrained by mTLS service identity; external releases require cryptographic consent
Unauthorized code deployment	A service image, model file, or manifest is changed to process data differently	Manifests, container images, downloaded files, and base AMI measurements are committed to transparency records and checked against attestation before key release
Approved-code failure	Approved VM software mishandles plaintext while keys are active	Isolated networking, mTLS between VMs, role separation between app and inference services, disabled shared LLM prefix caches, logging controls, audits, and 7-day key expiry bound exposure
External service retention	An external provider retains data after authorized release	Release paths require consent and can stop future sharing; already transmitted data remains subject to the external provider's policies and controls
Client compromise	Malware, OS compromise, or malicious platform update steals client keys before release	Server-side controls cannot replace client integrity; Bee mitigates with platform key stores, passphrase recovery options, attestation checks, and user controls

1.3 Security Properties by Boundary

Bee's strongest privacy claim is not that the system never processes plaintext. A personal AI must process plaintext to summarize conversations, maintain memory, run inference, and act on the user's behalf. The claim is that plaintext processing is constrained to measured software identities, with client-gated key release, short-lived server-processing keys, isolated networks, and auditable runtime records. The table below separates the positive property from the residual risk at each boundary.

Boundary	Security Property	Mechanism	Residual Risk
Client to Confidential VM	Server-processing keys are released only to a verified runtime identity	TLS certificate pinning, NitroTPM attestation, Sigstore inclusion proofs, PCR matching, certificate public-key binding	A compromised client can leak client-held secrets before release
Persistent storage	Canonical databases, queues, caches, and object stores are not intended to hold user plaintext	Envelope encryption under keys derived from client-generated account roots; scoped keys shared only after attestation; derived search caches are dropped when keys are definitively unavailable	Metadata such as account identifiers, timestamps, queue state, delivery status, and active derived-index state can remain visible
VM runtime identity	Material changes to the plaintext-processing runtime are externally visible	Base AMI PCRs, PCR14 manifest measurement, container digests, S3 artifact hashes, and transparency records	Approved code can still contain bugs or behavior that must be governed by product privacy review
Internal VM traffic	Plaintext moves only across intended service roles	mTLS between Confidential VMs, role-specific certificates, isolated networks, and explicit bridge configuration	Bee's current architecture is attested and access-controlled; stronger non-targetability remains a routing-hardening area
External integrations	Third-party releases require explicit authorization and are bounded going forward	Cryptographic consent records, expiry, revocation, provider scopes, and encrypted or low-content push channels	Data already released to a provider follows that provider's retention and deletion model
Operations	Operators should not be able to inspect or mutate live plaintext-processing hosts interactively	No SSH, SSM, EC2 Instance Connect, or shell path on Confidential VMs; replace-and-measure deployments; SCPs and separated duties	Process discipline, code review, and audits remain necessary for logging and approved-code behavior

1.4 Why Bee Uses a Stateful Privacy Model

Some private-processing designs are optimized for request-oriented AI features where a user explicitly asks for a response and the processing service can discard session state after completion. Bee's product model is different. Bee is a personal AI wearable and assistant intended to run in the background, ingest ongoing context, maintain memory, notice patterns, and proactively help the user with conversations, reminders, insights, summaries, and connected-service workflows. A useful agent of that kind cannot be fully stateless: it must be able to process new events after capture, connect them to prior context, and perform background work when the user is not actively holding an interactive session open.

Bee therefore makes statefulness explicit, purposeful, and bounded. Long-term content remains encrypted at rest under keys derived from client-generated account secrets. The server receives only scoped server-processing keys after the client verifies an attested Confidential VM. Those keys are kept in VM memory and Confidential Redis keystores for at most 7 days of inactivity so background processing can continue while the wearable and assistant operate. If the client stops refreshing keys, queued jobs that need plaintext fail closed because they cannot obtain a key. The result is a stateful design that matches the product requirement while keeping the decryption capability short-lived, measured, and revocable.

Bee has a different active-processing window than a stateless request processor. The benefit is that Bee can deliver proactive assistance instead of only responding inside a single request. The control set is designed around that tradeoff: client-controlled account roots, attestation-gated key release, transparency-recorded software identities, isolated Confidential VM networks, mTLS for internal service calls, ephemeral key caches with TTLs, cryptographic consent for external releases, and revocation paths for stopping future sharing.

2 Platform Components

Bee consists of these primary components that work together to provide a secure and private platform for AI applications.

1. **Client Apps**: Mobile, desktop, or CLI apps are the entry points for users to interact with Bee and are responsible for managing the user's keys.
2. **Hardware**: Bee works with the Bee wearable, keeping capture, pairing, and key-release behavior tied to an attestation-verifying client path.
3. **Confidential VMs**: Virtual machines that process encrypted user data and decrypt it only after receiving scoped keys from an attestation-verifying client. These VMs can prove that they are running the required software before they receive sensitive material.

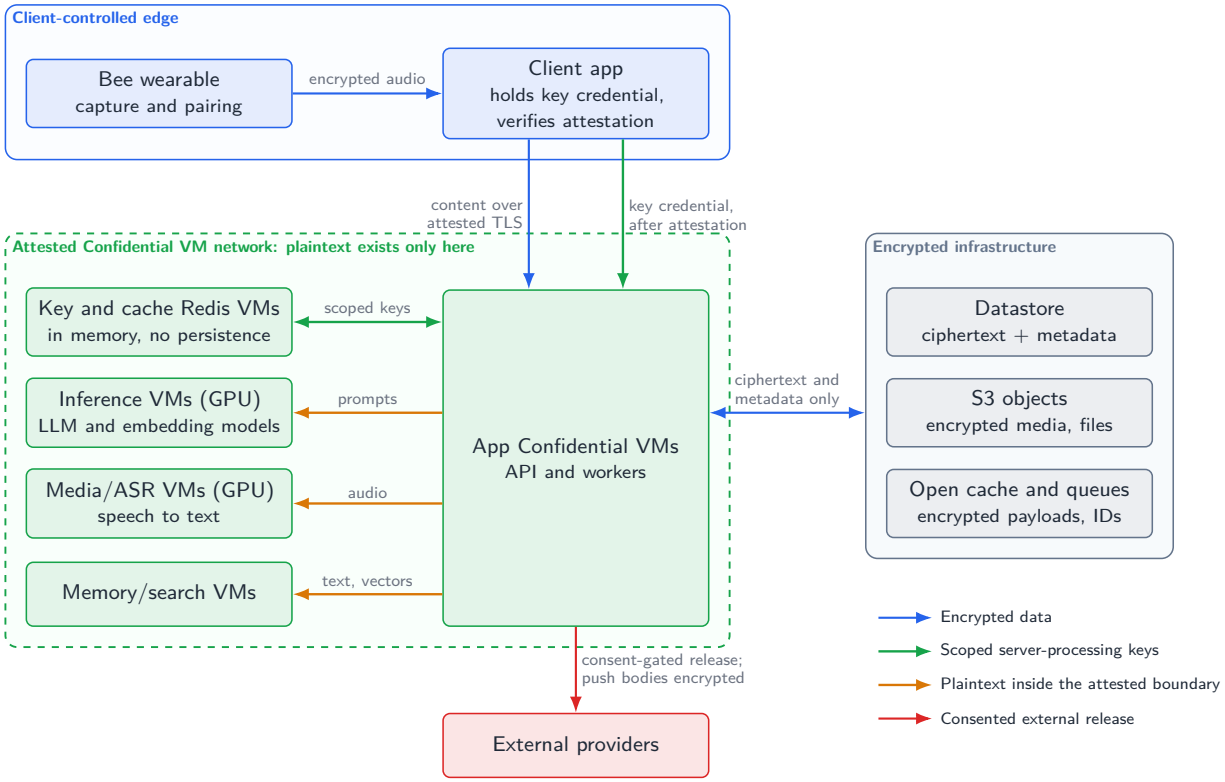

Figure 1: End-to-end privacy boundary. Clients send content and key credentials only to attested Confidential VMs. Plaintext processing, including inference, speech recognition, and search, stays inside the attested network. Persistent infrastructure outside the boundary receives ciphertext and metadata, and external releases are consent-gated.

3 Client Apps and Hardware

Client apps (mobile, desktop, or CLI) serve as the entry point for users and are responsible for managing encryption keys, verifying server attestations, and establishing secure connections to Confidential VMs. Hardware devices either connect through these apps or implement an equivalent attestation-verifying client path before any server-processing key is released.

The Bee wearable connects to the mobile app via Bluetooth. The mobile client is responsible for key management and must perform VM attestation verification before releasing keys or plaintext-bearing requests.

For the Bee wearable, audio traffic over Bluetooth is encrypted with an additional layer beyond standard Bluetooth AES encryption. Each Bee device contains a provisioned public key and accepts only commands signed by the corresponding backend signing service.

When a Bee device is paired, the mobile app generates a session encryption key for audio streaming. This key is sent to the backend signing service, which first verifies that the authenticated user owns an active Bee device record for the requested device identifier. Only then does the signing service sign the session material with the device's private signing key. The signed key is transmitted to the device, which verifies the signature against its embedded public key before enabling audio streaming. This protects the Bluetooth audio stream from passive observers and other apps that can observe Bluetooth traffic but cannot access the Bee app's key material, and it prevents a caller from requesting signed session material for an unregistered or other-user device.

The wearable signing key is separate from user-content encryption keys and does not grant access to stored content. Because Confidential VMs have finite lifetimes and software modifications are performed only through replacement rather than in-place updates, the device signing key is stored encrypted in persistent backend storage and unwrapped only inside the approved signing service. The signing service is included in the measured VM manifest, and per-device keys can be rotated or revoked without rotating user account secrets. The signing service is therefore part of the wearable trust boundary: an attacker would need to compromise the Bee app, the operating system, or the approved signing service to authorize an attacker-controlled session key.

3.1 User Capture Controls

Bee's stateful model is paired with user-visible capture controls. The client experience exposes listening state and capture controls at the point where the wearable is used. During onboarding, Bee starts listening only after the user enables capture. After setup, users can mute or stop listening from the paired phone UI. The phone UI reflects whether capture is muted and writes mute-state changes to the device over Bluetooth. For supported Bee firmware, press-to-record behavior is available: Bee does not listen by default, a device button starts and stops capture, a green light indicates active listening, and an auto-timeout can stop capture after a period of silence.

These controls sit below the server privacy architecture but are important to the complete model. Attestation and key release constrain where plaintext can be processed after capture; device and client controls constrain when capture begins and whether background collection continues. In the intended launch posture, user-visible capture controls, attested key release, and bounded server-side state operate together.

4 Confidential VMs

4.1 Overview

Bee deploys its servers as Docker containers running inside Confidential VMs. Each Confidential VM is a stripped-down Linux distribution containing only the hardened kernel, networking, and GPU drivers necessary for operation. There is no SSH, SSM, or shell access; operators do not have an interactive remote administration path into these machines. This follows the AWS guidance for isolating instance data from a customer's own operators: remove interactive access, include only trusted software, firewall the instance, and keep storage read-only[2].

Confidential VMs communicate with each other to distribute workloads across multiple nodes. Each VM receives its configuration from a userdata manifest. The built-in EC2 userdata loading mechanism is disabled and replaced with a custom implementation: the per-boot path no longer executes legacy shell userdata, and the manifest loader rejects legacy bash userdata instead of treating it as an executable payload. A custom measured boot tool downloads Docker containers and S3 files (such as model weights) specified in the manifest, then calculates hashes locally from the downloaded data. The tool hashes the Docker containers, downloaded files, and the manifest contents itself. This combined hash extends PCR14 before obtaining the NitroTPM[6] attestation document, ensuring that any change to the runtime configuration is reflected in the attestation.

After obtaining the attestation document, the VM loads the corresponding Sigstore proofs from an S3 bucket and cryptographically verifies that they match. Only after this verification does the VM request a certificate from a private Certificate Authority, with both the attestation document and Sigstore proofs embedded in the certificate. This certificate is then used for all TLS connections, both with clients and between VMs.

The VMs utilize NitroTPM for measured boot: UEFI firmware, the bootloader, and the kernel measure each boot stage into NitroTPM PCRs[6], and Bee's measured-boot tool extends PCR14 to carry that chain of trust through application startup. The NitroTPM attestation document is a CBOR-encoded COSE_Sign1 object containing PCR values, optional user data, and an AWS Nitro Attestation PKI certificate chain[7]; throughout this paper, "NitroTPM attestation document" refers to this artifact. AWS Nitro's minimized design removes many conventional hypervisor attack surfaces, and AWS has published formal-methods work for selected low-level boot and control-plane components[8][9]. Sigstore verification is performed completely offline using proofs stored in S3; the VMs never contact external Sigstore infrastructure.

Confidential VMs are created from a base AMI that is rebuilt periodically, combined with manifest files that define which workloads and data need to be present on each machine. The VMs have a read-only root filesystem with an overlayfs backed by RAM for any runtime modifications. The boot path accepts only expected local NVMe instance-store disks for application data. Those disks are mounted through an ephemeral LUKS mapping with an in-memory passphrase and detached header, so the decrypted mapping exists only during the current VM lifetime. These local disks are used primarily for Docker images and model weights. Confidential VMs cannot be rebooted; they can only be terminated. Once a VM is terminated, data on these disks is discarded with the instance.

All workload containers run in a separate Docker network, further isolating them from the VPC. The VM initializer assigns deterministic container IPs and exports per-container connectivity metadata so the AMI firewall can install `DOCKER-USER`, host `INPUT`, and host `OUTPUT` rules. Managed application containers default to no host-local, local-network, metadata-service, or external egress unless the manifest grants that direction explicitly. Expose containers, authentication sidecars, bridges, OpenTelemetry scrape targets, and declared internal links become one-way source or destination allow rules. All Confidential VM AMI flavors set Docker's default runtime to gVisor `runsc`. The runtime is fixed in the measured image and cannot be changed at deploy time. Confidential VMs connect to each other through predefined bridges specified in the manifest, implemented using HAProxy with an SPOE agent that verifies certificates. This architecture allows any TCP-compatible software running in Confidential VMs (such as Redis) to communicate securely without requiring each application to implement its own certificate verification.

All Confidential VMs are deployed to an isolated network without general internet egress. Network egress is allowed only to RDS for database storage, authenticated `rediss://` ElastiCache endpoints in isolated subnets, and tightly scoped Lambda functions for generic external integrations. Sensitive data is encrypted using customer keys before reaching non-confidential ElastiCache and RDS, and ElastiCache access is further limited by generated password users, user groups, and service-specific security groups. User-scoped push notifications are encrypted when the VM has the recipient's encryption context: push providers receive generic visible notification text plus an encrypted payload that the client notification extension decrypts locally. If an encrypted push cannot be prepared for a particular delivery path, Bee uses a minimal fallback notification rather than exposing rich user content. To upgrade code, new VMs are deployed with an updated manifest or base AMI, and existing VMs are terminated and replaced. Confidential VMs can ingest data from third-party services through the Lambda integrations. They release plaintext or provider-specific tokens to an external integration only when an active cryptographic consent record authorizes that specific data flow; otherwise outbound user content remains encrypted with customer-controlled keys.

A security agent runs at the kernel level within each Confidential VM for threat detection. This is Amazon-internal security software that has been stripped down to prohibit dynamic configuration. The agent is not exempt from the attestation model: the exact package, configuration, and kernel integration are included in the VM manifest and PCR measurements, and any update changes the attested VM identity. The agent is also subject to the same network egress policy as the rest of the VM. Because the agent source is not available to the Bee team, Bee treats it as a pinned runtime dependency rather than an invisible exception: its measured binary identity is fixed, dynamic remote configuration is disabled, egress is limited, and the agent must appear in the transparency-recorded configuration.

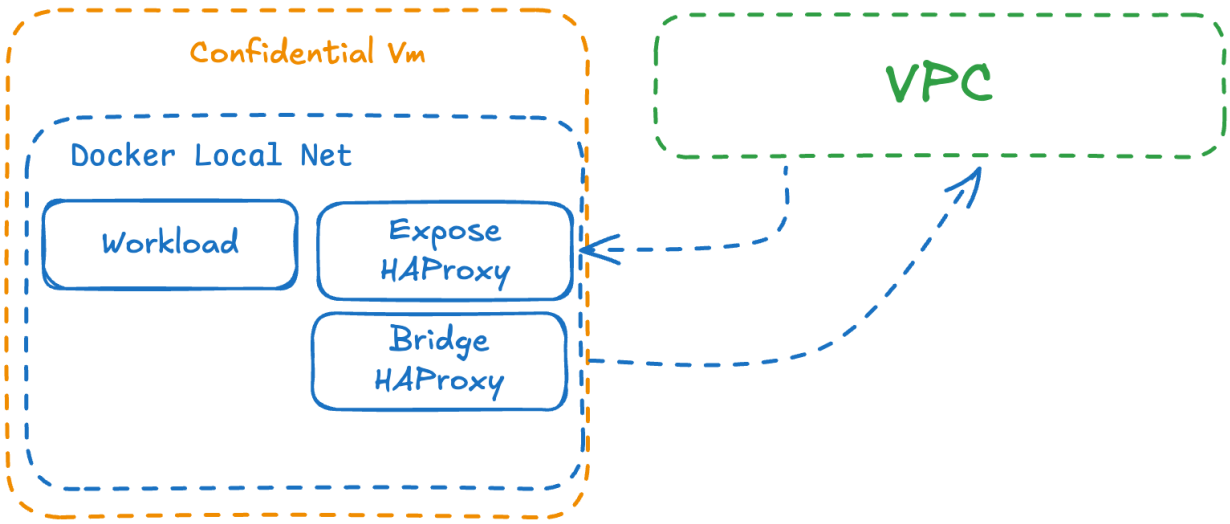

Figure 2: Confidential VMs.

4.2 Working with keys

Multiple dedicated Redis instances running inside Confidential VMs hold scoped server-processing keys in memory for resilience. Redis persistence is disabled for these key stores. Access to key Redis, shared Redis, the search index, and related Confidential VM services is gated by mTLS plus Sigstore service-identity policy, so only the roles declared in the measured manifest can reach the corresponding service. Each application server continuously synchronizes active keys with the Redis instances. When a client connects to the server, it shares the relevant server-processing key, which the server then uses to decrypt data when needed. When a client runs an HTTP request, the server fetches the key from Redis if it is not already in local memory and decrypts the data on the fly before returning it to the client. When background tasks are needed, the server resolves the key the same way to decrypt and encrypt data.

4.3 Routing and Targetability

Bee optimizes the routing layer for attested plaintext containment rather than anonymous request routing. Requests to public Bee API endpoints are routed through Bee-controlled load balancing into public-facing Confidential VM roles, and internal traffic then moves over configured VM-to-VM bridges. The privacy goal of this layer is to ensure that user plaintext and server-processing keys are available only to attested runtimes, while standard load balancers, databases, caches, and operators remain outside the plaintext path.

The routing controls are central to that goal. Public TLS is enabled only for roles that intentionally accept client traffic. Internal services use mTLS by default, and bridge definitions in the manifest define which internal service names can be reached. The CDK deployment model places Confidential VMs in private isolated subnets unless a service explicitly opts into internet egress, and production services use explicit connectivity maps rather than broad shared security-group reachability. Application roles bridge only to named Redis, media, model, and search services. Security groups allow DNS, S3 artifact access, local HTTPS, RDS, authenticated ElastiCache, and selected peered VPC traffic rather than unrestricted outbound access. These controls materially reduce arbitrary exfiltration paths and make service-to-service reachability part of the measured, reviewable configuration.

The main routing hardening area is non-targetability. An operator who can influence load-balancer routing or service placement may be able to send a particular user to a particular approved VM instance. Because clients verify the attested software identity rather than the operator's routing decision, this does not by itself reveal plaintext to non-attested systems. Bee's current property is therefore best described as attested-processing containment, with per-user routing transparency, public attestable instance reports, and stronger non-targetability controls as future hardening areas.

4.4 Inference and GPU Boundary

Bee separates application servers from inference services. Application servers hold server-processing keys and are responsible for decrypting content, enforcing consent, and deciding what plaintext is sent for inference. Inference services receive only the plaintext prompt or embedding input required for a specific operation over mTLS from an application server. They do not receive raw account secrets.

Model-serving containers, GPU drivers, model weights, and downloaded model files are part of the measured runtime configuration. A change to a model image, model-weight S3 object, GPU-serving container, or manifest changes the VM's measured identity and therefore changes the attestation and transparency record clients must accept before releasing keys. This gives users and auditors a path to verify which inference stack was authorized to process data.

This separation gives Bee a narrow and reviewable inference boundary. Prompt content is plaintext while inference runs, so the application layer controls what context is sent and the inference layer controls only the model operation it receives. Shared model-serving backends are configured without shared prefix/radix caches where cross-user cache reuse would create a privacy boundary concern; the system favors recomputation over retaining prompt-derived prefixes across users. Bee relies on narrow routing, mTLS, measured artifacts, output handling, logging discipline, and audits to keep that boundary explicit. GPU memory sits inside the instance's Nitro isolation and no-operator-access boundary; Bee does not rely on GPU-specific confidential-computing features. This paper focuses on Bee's controls around what reaches inference and how those inference components are measured.

5 Attestation and Transparency Logs

The security of Confidential VMs depends on the ability to verify that only approved software runs inside them. This verification relies on two complementary mechanisms: NitroTPM attestation provides proof, rooted in the Nitro System, of what code and configuration booted on an EC2 instance[1][6], while Sigstore Rekor provides an append-only, publicly auditable transparency log for signed software metadata[10][11].

5.1 Base AMI Build Process

Base AMIs are built as hardened Linux images configured for NitroTPM support and measured boot. AMIs are updated periodically to upgrade the kernel, drivers, and security patches. All base AMIs are persisted indefinitely, enabling retrospective audits of any software version that has ever processed user data.

During the AMI build process, expected PCR values are computed offline by hashing the boot components (kernel, initramfs, and bootloader) without actually booting the image. These base AMI PCR values are persisted for audit purposes but are not committed to Sigstore directly; a manifest is always required to establish a valid attestation chain. At runtime, the boot chain extends the same measurements into NitroTPM during the boot process, producing PCR values that must match the precomputed values. These verified PCR values are then used as the foundation for subsequent stages of the boot sequence, establishing a chain of trust from the base system through to application startup.

5.2 Runtime Configuration Measurement

Beyond the base AMI, each Confidential VM runs specific workloads defined by a manifest. The custom measured boot tool downloads Docker containers and S3 files specified in the manifest, calculates hashes of the downloaded content, and combines these with the manifest hash. This combined hash extends PCR14, ensuring the attestation document reflects both the base system and the specific runtime configuration.

Before deployment, when a new configuration is detected, a Sigstore proof is produced through the following process: the manifest is uploaded to the transparency service, which verifies that the referenced base AMI exists and is persisted in cold storage. The transparency service also maintains replicas of ECR and S3, allowing it to verify that all required artifacts (Docker images, model weights, and other files) are persisted and available for audit. The transparency service then parses the manifest file and recomputes the expected PCR values. The final Sigstore commitment publishes PCR4, PCR7, and PCR12 (derived from the base AMI) along with PCR14 (derived from the manifest). If all values match, the proof is committed to the

public Sigstore Rekor log. The public-good Rekor instance is operated as part of Sigstore/OpenSSF rather than by Amazon, and its append-only behavior is independently monitorable[10][11].

After the proof is committed, the Sigstore proofs are uploaded to an S3 bucket that is replicated to each region, making them accessible from isolated networks without internet egress. When a VM boots, it downloads the proofs and verifies their integrity before proceeding. This verification serves as a fail-fast mechanism to detect infrastructure problems early, providing an additional layer of defense in depth. The VM then checks that the proofs match its PCR values before requesting a certificate. Even without this early check, clients would reject connections to VMs with mismatched attestations, but failing fast at boot time prevents wasted resources and surfaces configuration issues immediately.

5.3 Sigstore Proof Structure

Sigstore proofs use a certificate chain to establish trust. The chain consists of a root certificate for signing proofs, an intermediate certificate, and a leaf certificate. The relevant signing certificate and signed payload metadata are committed to the Sigstore transparency log, and the Sigstore proofs object includes the inclusion material needed by the client. The root certificate is hardcoded in client code, eliminating the need to fetch it at runtime.

The signing process publishes two items to Sigstore: the leaf signing certificate, verified against the intermediate CA, and the AMI payload (canonical JSON containing the PCR map and metadata) signed by an ephemeral leaf key using ECDSA P-256 with SHA-256. The AMI payload signature is countersigned by an RFC 3161 timestamp authority, creating a verifiable chain: transparency log $\rightarrow$ timestamp authority $\rightarrow$ leaf certificate $\rightarrow$ intermediate CA.

The complete Sigstore proofs object contains three proofs: the leaf certificate inclusion proof, the VM configuration signature proof, and the TSA time proof. All three are embedded in the TLS certificate as X.509 extensions, allowing clients to verify the complete chain of trust during the TLS handshake without contacting external services.

6 TLS with Attestation

When a Confidential VM starts, it obtains a NitroTPM attestation document containing PCR values that represent cryptographic measurements of the boot process, operating system, and application code. The VM fetches the corresponding Sigstore proofs from an S3 bucket and verifies that the PCR values match the published records. After successful verification, the VM requests a certificate from a private Certificate Authority. The CA issues a certificate only to VMs that present valid, matching attestation and Sigstore proofs.

6.1 TLS Modes

The system supports two TLS modes, each serving different security requirements.

Public TLS is accessible from the internet and uses a certificate from the private CA. In this mode, only the client verifies the server's integrity; the server authenticates the user at the application layer rather than through a client certificate. User devices connect using this mode. When a client connects, it verifies the certificate chain, extracts the embedded attestation and Sigstore proofs, and independently validates them. The client checks that PCR values match transparency log entries and that the attestation document is signed by a certificate chain rooted in the AWS Nitro Attestation PKI, proving it was issued by the AWS Nitro System. Only after all checks pass does the client establish the connection and share account keys. If any measured component has been modified, the PCR values change and the client refuses to connect.

This mode is enabled only on specific roles and marked as a high-risk public-exposure option in configuration. Multiple layers of checks prevent accidental enablement: the choice is embedded in the base AMI configuration, in deployment scripts, and within the manifest. Only a few specific places allow enabling it, making accidental activation difficult and reviewable.

Mutual TLS (mTLS) requires both sides to present valid certificates to establish a connection. This is the default mode, used to transfer data between application servers and GPU servers, or to Confidential Redis instances for caching results. During boot, depending on the manifest, not every node receives a client

certificate. Software that is not supposed to have outgoing connections does not receive a client certificate and cannot connect to other workloads.

Servers are logically split into two categories based on their access to sensitive material. Application servers have access to server-processing keys, receive mTLS client certificates, and can connect to other VMs directly. Inference servers do not have access to raw customer keys; they receive only the plaintext inputs required for a specific inference request over mTLS from application servers. This separation reduces blast radius. Combined with the isolated network architecture, it limits practical exfiltration paths from an inference server and keeps application-level output filtering, logging controls, and runtime monitoring focused on the boundary where plaintext is intentionally used.

6.2 Certificate Attestation Extension

Attestation data is embedded in the TLS certificate using a custom X.509 extension with OID 2.23.133.5.4.9 (TCG DICE Conceptual Message Wrapper). The extension contains a CBOR-encoded CMW Collection with the following structure:

- **pubkey-hash**: SHA-256 hash of the certificate's public key, binding the attestation to this specific certificate
- **pcr-selection**: List of PCR indices included in the attestation (4, 7, 12, 14)
- **nitrotpm**: The raw AWS NitroTPM attestation document (CBOR/COSE_Sign1)
- **sigstore**: Gzip-compressed JSON containing the Sigstore bundle

The Sigstore bundle is stored as gzip-compressed JSON rather than raw CBOR due to size constraints; the full Sigstore proof structure in CBOR format would exceed practical X.509 certificate size limits. The JSON structure follows this schema:

```
{
  "version": 1,
  "expiry": 1735689600,
  "context": { "ami": "ami-abc123", "manifest": "..." },
  "pcrMap": {
    "PCR4": "a1b2c3...",
    "PCR7": "d4e5f6...",
    "PCR12": "789abc...",
    "PCR14": "def012..."
  },
  "caChain": "-----BEGIN CERTIFICATE-----\n...",
  "amiInclusionProofs": [{
    "logIndex": "12345",
    "logId": { "keyId": "base64..." },
    "inclusionProof": {
      "logIndex": "12345",
      "rootHash": "base64...",
      "treeSize": "67890",
      "hashes": ["base64...", "..."],
      "checkpoint": { "envelope": "rekor.sigstore.dev\n..." }
    },
    "canonicalizedBody": "base64..."
  }],
  "signingCertificateEntries": [{
    "certificateBytes": "base64...",
    "inclusionProofs": [...]
  }],
```

```
  "timestamp": ["base64-rfc3161-response..."]
}
```

During certificate generation, the VM computes the SHA-256 hash of the certificate's public key and passes it as user data to the NitroTPM attestation request. This binds the attestation document to the specific certificate being issued. The CMW Collection is then embedded in the CSR, and the Private CA issues the certificate using a passthrough template that preserves the custom extension.

6.3 Attestation Verification

Client verification proceeds through a series of cryptographic checks that bind together the NitroTPM attestation, Sigstore proofs, and the TLS certificate itself. Attestation verification only begins after standard TLS certificate chain validation succeeds. The TLS certificate must first be verified against the Private CA root certificate (also hardcoded in the client). Only after the TLS handshake establishes a valid certificate chain does the client proceed to verify the embedded attestation extension.

First, the client extracts the CMW Collection from the X.509 extension and verifies the `pubkey-hash` matches the SHA-256 hash of the certificate's public key. This confirms the attestation was created specifically for this certificate and prevents replay attacks using attestations from other certificates.

Next, the client parses the NitroTPM COSE_Sign1 attestation document and extracts the PCR values. The `user_data` field in the NitroTPM attestation must contain the hexadecimal representation of the certificate's public key hash, providing a second binding between the attestation and certificate.

The client then decompresses the Sigstore JSON bundle and performs the following validations:

Expiry check: The `expiry` field must be greater than the current Unix timestamp. If the proof has expired, verification fails immediately.

Context policy check: The authenticated `context` field must satisfy the policy expected by the caller. For client key release this includes the expected Bee owner, deployment environment, and key-release allowance such as `allowKeys=true`; for internal service traffic, Sigstore context is also used as service identity for role-scoped mTLS policy. Missing, malformed, or unexpected context values fail verification.

PCR matching: Each entry in `pcrMap` (PCR4, PCR7, PCR12, PCR14) is compared against the corresponding value from the NitroTPM attestation. All values must match exactly.

AMI proof reconstruction: The client reconstructs the canonical AMI payload by combining `context`, `expiry`, `pcrMap`, and `version` into a JSON object with NFKC-normalized Unicode strings. The SHA-256 hash of this canonical form must match the digest in each `amiInclusionProofs` entry's `canonicalizedBody`. The AMI Rekor entry signature is verified and the signing key is bound to a certificate chain rooted in the pinned Bee AMI-builder trust root.

Rekor inclusion verification: For each inclusion proof, the client verifies the Merkle tree path from the entry to the signed checkpoint. The checkpoint signature is verified against the trusted Rekor public key (embedded in client code). The checkpoint origin must match the trusted Rekor log URL.

Certificate chain verification: The signing certificate root CA (PceAmiBuilderRootCa) is hardcoded in the client binary. The `caChain` field contains intermediate certificates, and the client verifies that this chain includes the hardcoded root. The ephemeral signing certificate from `signingCertificateEntries` must verify against this pinned root through the intermediate chain. The verification time is derived from the RFC 3161 timestamp, not the current time, allowing certificates to remain valid even after their notAfter date as long as the timestamp proves they were signed while valid.

Timestamp verification: The RFC 3161 timestamp response is parsed and verified against trusted timestamp authorities from the Sigstore trusted root. The timestamp's signed attributes, including `messageDigest`, must match the TSTInfo bytes, and the timestamp signature itself must verify. The timestamp's message imprint must match the SHA-256 hash of a reconstructed "SignablePayload" containing the digest, signature, and public key from the AMI inclusion proof. This binds the timestamp to the specific signed content.

Signing certificate proof: The TBS (To-Be-Signed) portion of each signing certificate is hashed with SHA-256, and this digest must match the one recorded in the corresponding `signingCertificateEntries` inclusion proof.

NitroTPM chain verification: The NitroTPM attestation document contains a certificate chain signed by the AWS Nitro Attestation PKI[7]. The client verifies the COSE_Sign1 protected headers and signature

over the attestation payload, then verifies the certificate chain against the pinned AWS Nitro root certificate. The verification uses the leaf certificate's notBefore time as the reference point, ensuring the chain was valid when the attestation was created; freshness is provided by the fact that the attestation is generated at boot and bound to a certificate whose lifetime is bounded by the private CA. This proves the attestation originated from the AWS Nitro System.

Time-based verification: All certificate chain and inclusion proof validations use the RFC 3161 timestamp as the verification time, not the current time. This is critical because Sigstore proofs may be used long after they were created. Ephemeral signing certificates are short-lived by design, so verifying against current time would cause valid proofs to fail once the certificate expires. By using the timestamp, the client confirms that at the moment of signing, all certificates were valid and the proofs were correctly committed to the transparency log.

Only after all checks pass does the client consider the connection trusted and proceed to share encryption keys. Mobile clients enforce this verification before server-processing keys are released.

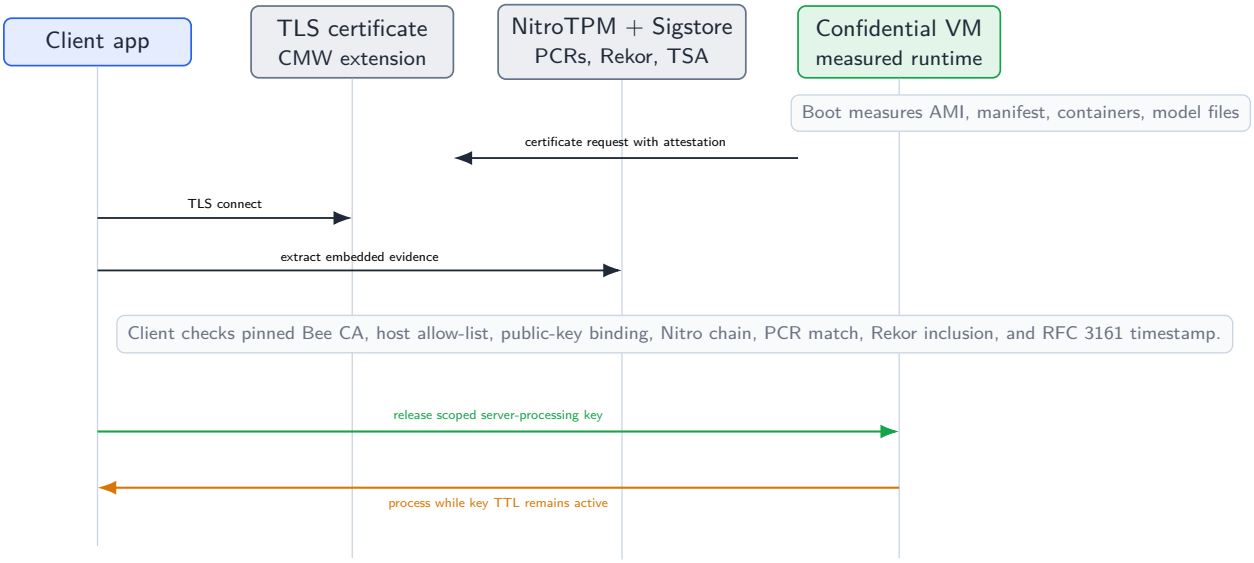

Figure 3: Attestation-gated key release. The client releases server-processing keys only after the TLS certificate, NitroTPM attestation, and Sigstore transparency evidence all verify against pinned trust roots.

6.4 User-Facing Transparency

The attestation and Sigstore data is carried in the TLS certificate so clients can verify the VM identity during connection establishment. The iOS client includes certificate pinning to the Bee CA, host allow-listing for Bee domains, attestation extension parsing, public-key binding checks, PCR matching against the Sigstore bundle, inclusion proof validation, and pinned trust roots for the relevant verification chain. The production privacy model requires this verification to be enforced before the client releases server-processing keys.

The same data can support user-facing and auditor-facing transparency reports. The current architecture makes the verification evidence available to clients and auditors with access to the certificate, manifests, artifact archives, and transparency records.

6.5 Attestation Report Contents

A useful transparency report should show both the cryptographic evidence and the client decision that followed from that evidence. Bee's report format can be derived from data already present in the attested TLS connection and transparency artifacts.

Report Field	Meaning	Why It Matters
Service domain and role	Public hostname, VM role, and module identifier	Shows which service received keys or plaintext
Certificate identity	Bee CA chain, certificate public-key hash, and attestation public-key binding	Prevents replay of attestations across certificates or hosts
NitroTPM measurements	PCR4, PCR7, PCR12, PCR14, and attestation freshness	Binds the live VM to measured boot and runtime configuration
Runtime manifest	Manifest hash, base AMI identifier, container image digests, S3 artifact hashes, model-file hashes, and bridge configuration	Identifies the exact approved software and artifacts that formed the runtime
Transparency proof	Rekor log index, log ID, root hash, checkpoint, inclusion path, signing certificate entry, and RFC 3161 timestamp	Shows the approved measurement was committed to an append-only public log before use
Client decision	Verification time, accepted or rejected result, and failure reason when rejected	Converts raw evidence into an auditable statement about whether keys were released

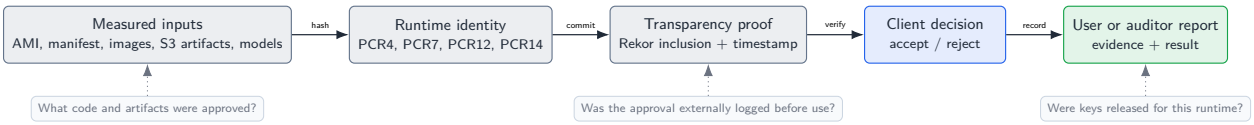

Figure 4: Transparency report construction. The report links measured deployment inputs to public transparency evidence and the client-side key-release decision.

This report is a verifiability artifact, not an execution transcript. It can prove what measured software identity the client accepted, which artifacts were covered by the manifest, and whether the client released server-processing keys. It does not prove the absence of bugs in approved code, and it does not replace source review, runtime monitoring, or product-level privacy review. Its value is that disputes can be grounded in concrete cryptographic evidence rather than internal assertions.

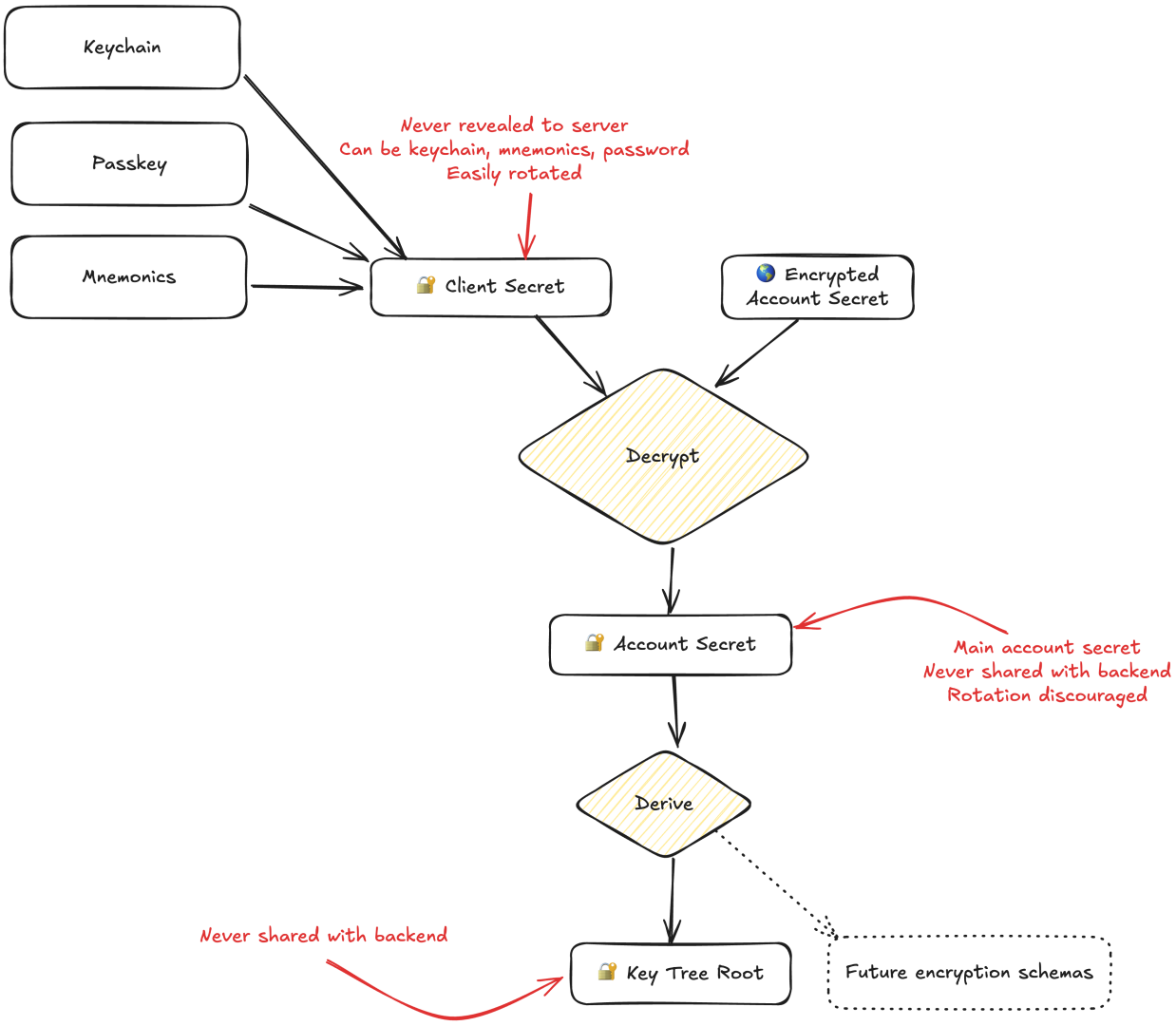

Figure 5: Getting Key Tree Root

7 Encryption

Bee's encrypted storage path uses symmetric key cryptography with keys derived from client-generated root secrets. The user's device holds a Client Secret, which decrypts an Account Secret stored on the server only in encrypted form. From the Account Secret, a hierarchical key tree derives purpose-specific encryption keys. Content is encrypted using envelope encryption: a random Data Encryption Key encrypts the content, and a derived key encrypts the DEK. Scoped server-processing keys are shared with Confidential VMs only after attestation verification, and expire after 7 days of inactivity.

7.1 Sharing keys with a Confidential VM

The client **never** shares server-processing keys without first verifying that the connection is established with a trusted Confidential VM through attestation verification. Keys are shared with API requests to the Confidential VM. Upon receiving the keys, the application server replicates them to ephemeral Redis instances, which are also Confidential VMs running in the same isolated network. Other application servers then replicate these keys from Redis to their local caches.

Background jobs that need to process user data attempt to fetch the user's key from Redis or from their

local cache. If the key is not available, the job aborts execution because it cannot proceed without the ability to decrypt user data. This ensures that processing only occurs when the user has actively shared their keys.

Long-term server-processing keys have a maximum TTL of **7 days**. During replication, the TTL is stored as an absolute Unix timestamp of expiration and propagated to all Redis instances and application server caches. Each client request resets this expiration timestamp, extending the TTL. If no client activity occurs for 7 days, the keys expire from all key caches within a bounded window determined by clock synchronization and cache cleanup intervals, leaving account data encrypted at rest. The backend cannot decrypt user content until the client shares the relevant keys again.

7.2 Client Secret

The Client Secret, denoted by S_c, is the starting point for constructing cryptographic keys for user data. The Client Secret is deterministic key material that Bee cannot access and cannot recover if the user loses possession of it. The system is flexible about the source of this secret: a **passkey**, a **platform-specific key** (such as Apple Keychain), or a **mnemonic phrase** similar to the seed phrase in crypto wallets. The default choice is a **platform-specific key** stored and managed by the mobile platform.

7.3 Account Secret

Bee stores an Encrypted Account Secret, denoted by E_a, in the database. It is generated by the client during account creation or Client Secret rotation. Bee can store multiple encrypted account secrets for the same account so users can use multiple client-side recovery or key-management methods.

After authentication, the client fetches E_a from the backend and decrypts it using S_c with **AES-256-GCM**. If there are no encrypted account secrets, the client generates a random 64-byte string, encrypts it using S_c, and stores the encrypted value in the database. We denote this plaintext account secret as M.

7.4 Hierarchical Key Derivation

At this point, the client has a strong secret M that can be used to derive purpose-specific keys. To represent key derivation paths, we use path-like notation: $M/client/conversation/2026$, where M is the notation of the master key, and everything else is a key path. We use a scheme similar to Hierarchical Deterministic Wallets[12], but with a simpler derivation scheme that is not tied to a specific encryption or signing algorithm.

To start a Key Tree, we need to derive a root state from the master secret using **HMAC-SHA512**:

$$t_0 = \text{HMAC-SHA512}(M, \text{"}Bee\ Account\ Master\ Seed\text{"})$$

$$K_0 = t_0[0:32], C_0 = t_0[32:64]$$

K_0 is the root key material for the current node. C_0 is the root chain code, which is used as the HMAC key when deriving child nodes. For each subsequent level, K_N is the derived key material for that path and C_N is the chain code used to derive the next child.

To derive the next level of the key tree for subtree i (arbitrary text string) with a prepended zero byte, we need to use the following formula:

$$t_N = \text{HMAC-SHA512}(key = C_{N-1}, data = \text{concat}([0x00], i))$$

$$K_N = t_N[0:32], C_N = t_N[32:64]$$

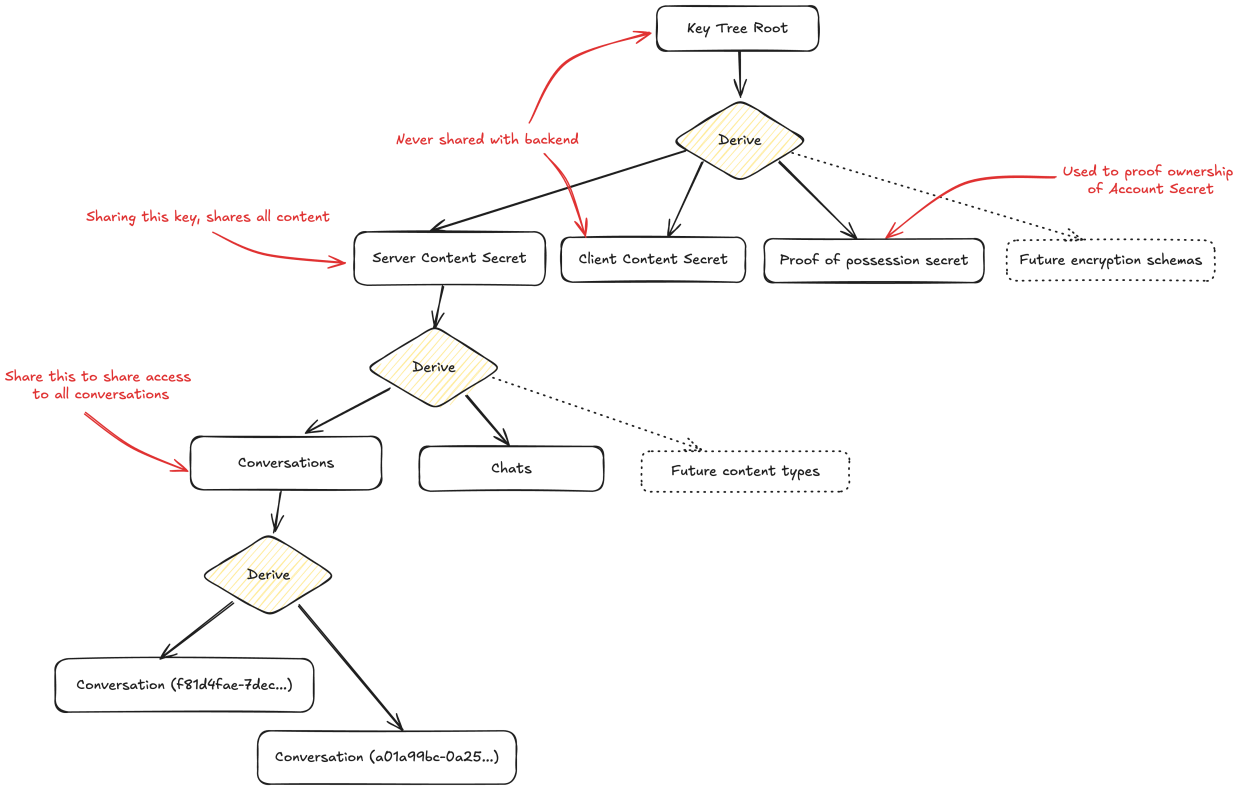

Figure 6: Hierarchical Keys

7.5 Key Tree

After building the key tree, the client has the ability to derive 32-byte keys for all purposes needed for its functionality.

$$S = \text{Derive}(\text{"}M/client/conversation/2026\text{"})$$

- $M/client$: This key tree is used to derive keys for client-side encryption that **never** leaves the device.
- $M/server$: This key tree is used to derive scoped server-processing keys that can be shared with attested Confidential VMs. Non-confidential backend stores receive only encrypted content and metadata.
- $M/account$: This is a dedicated key that is used to calculate the proof of possession of the account.

7.6 Proof of Possession

To verify ownership of the account, Bee uses a proof of possession, which is a **HMAC-SHA512** authentication tag calculated using the $M/account$ key. This provides a simple way to verify that the reconstructed key is correct. The $M/account$ path is deliberately separate from encryption key derivation paths to avoid any risk of key leakage through a hash of the actual encryption key material. This design proves possession of the account key material; it does not prove that a client is otherwise honest.

7.7 Content Encryption

The system uses symmetric key cryptography throughout, avoiding asymmetric encryption where possible. Content is encrypted using an envelope encryption scheme: we first generate a random Data Encryption Key

(DEK), then derive a key for the content and encrypt the DEK with it. Then we encrypt the content with the DEK.

Algorithm 1: Content Encryption

$C \leftarrow$ serialize_content() `// Serialize content to byte string`
$S \leftarrow$ load_customer_key() `// Load customer key from ephemeral storage`
$P \leftarrow$ generate_content_key_path() `// Generate a new content key path`
$A \leftarrow$ canonical_aad($P, metadata$) `// Authenticate version, content type, account, and path`

`// Prepare Keys`
$K_{dek} \leftarrow secure\_random(32)$ `// Generate a new Data Encryption Key`
$K_c \leftarrow$ Derive(S, P) `// Derive a key for the content`
$IV_{dek} \leftarrow secure\_random(12)$ `// Generate a new IV for the Data Encryption Key`
$IV_c \leftarrow secure\_random(12)$ `// Generate a new IV for the content`

`// Encrypt`
$E_{dek} \leftarrow IV_{dek} \parallel$ AES-256-GCM($key = K_c, data = K_{dek}, aad = A, iv = IV_{dek}$);
$E_c \leftarrow IV_c \parallel$ AES-256-GCM($key = K_{dek}, data = C, aad = A, iv = IV_c$);

`// Store`
persist($sec\_encryption\_key, E_{dek}$);
persist($sec\_encryption\_path, P$);
persist($sec\_aad\_metadata, metadata$);
persist($sec\_content, E_c$);

Algorithm 2: Content Decryption

$S \leftarrow$ load_customer_key() `// Load customer key from ephemeral storage`
$P \leftarrow$ load(sec_encryption_path) `// Load content key path`
$metadata \leftarrow$ load(sec_aad_metadata) `// Load authenticated metadata`
$A \leftarrow$ canonical_aad($P, metadata$);
$E_{dek} \leftarrow$ load(sec_encryption_key) `// Load encrypted Data Encryption Key`
$E_c \leftarrow$ load(sec_content) `// Load encrypted content`
$K_c \leftarrow$ Derive(S, P) `// Re-derive key for the encrypted DEK`

`// Decrypt`
$K_{dek} \leftarrow$ AES-256-GCM($key = K_c, data = E_{dek}[12:], aad = A, iv = E_{dek}[0:12]$);
$C \leftarrow$ AES-256-GCM($key = K_{dek}, data = E_c[12:], aad = A, iv = E_c[0:12]$);

`// Return`
return C;

AES-GCM IVs are generated randomly and must never be reused with the same key. The authenticated metadata includes at least the encryption version, content type, account identifier, and key path so that ciphertext cannot be replayed under a different interpretation.

7.7.1 Content Key Paths

For most content types, Bee derives a new key path. At the moment, all key derivation paths follow $M/server/\{content\}/\{uuid\}$. Here $\{content\}$ is a fixed string depending on the content type, and $\{uuid\}$ is a globally unique, securely generated UUID with stripped dashes and lower-case encoding. Bee explicitly stores this path in the database so it can rotate or change the content key path in the future. In practice, because the non-confidential backend does not have access to the Account Secret directly, Bee stores only the suffix of the path: $\{content\}/\{uuid\}$.

7.8 Key Rotation

Bee supports key rotation to limit the impact of compromised key material. When key rotation is needed, the preferred order is Client Secret, then Data Encryption Key, then Account Secret.

- **Client Secret**: This secret is in the user's possession and can be compromised if the user's device or recovery material is compromised. It can be rotated at any time by the client, and an account can have multiple Client Secrets active at the same time. Creating a new Client Secret creates a new Encrypted Account Secret and stores it in the backend.
- **Account Secret**: This is the main account secret used to derive other keys. It is never revealed to the non-confidential backend. It can be rotated, but not instantly, because existing content keys must be re-derived or re-wrapped.
- **Content Encryption Key**: This key is directly derived from the Account Secret and cannot be rotated separately. Changing it requires changing the derivation path or rotating the Account Secret.
- **Data Encryption Key**: This key encrypts content and can be rotated at any time, but rotation requires access to the old key and a currently available client or server-processing key.

7.9 Key Revocation and Failure Modes

Key availability is an explicit authorization signal. If a user stops using Bee, signs out, deletes local client secrets, or otherwise stops refreshing server-processing keys, the backend loses the ability to decrypt that user's content after the active key window expires. Application servers and background workers first check local memory and then Confidential Redis keystores. If no valid key is available, they abort work instead of falling back to a non-confidential decrypt path.

Revocation has several layers. Client Secret rotation creates a new encrypted account-secret enrollment and can remove old client-side recovery methods. Account Secret rotation is heavier because existing content keys must be re-derived or re-wrapped. Data Encryption Keys can be rotated when the old and new wrapping keys are both available. Consent revocation deletes the signed consent record so future external releases fail verification. These operations are forward-looking controls: they stop future processing or release, while already completed plaintext processing and already transmitted third-party data follow the relevant runtime or provider retention model.

The 7-day server-processing key TTL makes active access bounded rather than permanent. A key that has already been released to an attested VM can remain in local memory and Confidential Redis until expiration or cache cleanup. Derived search caches follow the same authorization signal: when the keystore can definitively determine that a user's server-processing key is no longer available, the search cleanup path removes that user's cached search-index collections rather than preserving plaintext-derived search material outside the active key window. Sensitive revocation paths can further tighten this window with explicit key deletion or cache invalidation where implemented, while the baseline privacy semantics remain simple: without periodic client refresh, server-side decryption capability expires.

7.10 CLI and Third-Party App Pairing

Users can extend their Bee account to CLI tools and third-party applications through a secure pairing process that grants a server-side capability without exposing the Account Secret. The pairing flow uses NaCl box encryption (Curve25519-XSalsa20-Poly1305) to establish a secure channel, ensuring that even if the pairing response is intercepted, only the intended recipient can decrypt the resulting bearer credential.

7.10.1 Pairing Flow

1. **Key Generation**: The CLI generates an ephemeral Curve25519 keypair using `nacl.box.keyPair()`. The public key is 32 bytes.
2. **Pairing Request**: The CLI sends a POST request to `/apps/pairing/request` containing:

- `app_id`: A registered application identifier
- `publicKey`: Base64-encoded Curve25519 public key

The server returns a `requestId` (derived from the public key) and an expiration timestamp (5 minutes).

3. **User Approval**: The CLI displays a QR code or URL (`https://bee.computer/connect#\{requestId\}`) that the user scans with the mobile app. The mobile app retrieves pairing request metadata and displays a confirmation screen showing the application name and a disclosure that the paired application will be able to access the user's Bee data.

4. **Token Generation**: Upon user approval, the mobile app creates a grant for the requested application. The Account Secret M is never exposed to the CLI or third-party applications. Instead, the mobile app re-encrypts the $M/server$ capability using an encryption key that only approved Confidential VMs can use after attestation. The backend stores this encrypted grant and generates a developer token that references it. Developer tokens are not currently permission-scoped: a paired application can access the same server-processable data as the user's own client, and the pairing disclosure states this. When the CLI makes API requests, the Confidential VM validates the token, decrypts the grant inside the VM, performs the necessary operations, and returns results. The CLI never has direct access to the underlying account key material, and the user can revoke the grant to disable the token. The server encrypts this token response using NaCl box with the following structure:

$$E = [v \parallel N \parallel K_e \parallel C]$$

Where v is a 1-byte version, N is a 24-byte random nonce, K_e is the server's ephemeral 32-byte public key, and C is the ciphertext.

5. **Token Retrieval**: The CLI polls the pairing endpoint until completion, then decrypts the token:

$$T = \text{nacl.box.open}(C, N, K_e, S_{cli})$$

Where S_{cli} is the CLI's secret key. The decrypted token is stored in the OS keychain.

7.10.2 Security Properties

- **5-minute expiry**: All pairing requests expire automatically, limiting the window for attacks.
- **One-time retrieval**: Pairing records are deleted after successful token retrieval, preventing replay.
- **Ephemeral keys**: Each pairing session uses fresh keypairs; compromising one session does not affect others.
- **Asymmetric encryption**: The CLI private key never leaves the CLI during pairing; the server only sees the CLI public key.
- **User verification**: The mobile app displays the requesting application details before approval, preventing unauthorized pairing.
- **Full-account access**: Paired-app tokens are not permission-scoped today. The protections are user approval with explicit disclosure, attested-VM-only mediation, and revocation, not per-scope authorization.

7.10.3 Token Storage

After pairing, the developer token is stored in the OS keychain (macOS Keychain, Linux libsecret, Windows Credential Manager). A file-based fallback to `~/.bee/token-{env}` with mode 0600 is used when the keychain is unavailable. The token is a sensitive bearer credential and is used in the `Authorization: Bearer` header for subsequent API requests to Confidential VMs.

8 Cryptographic Consent

Any sharing of user data with external services requires cryptographic consent, which is bound to keys derived from the user's Account Secret. When a user consents to share their data, Bee records a signed consent proof that specifies what data can be shared, with which service, and under what conditions. The consent payload is constructed by the service and signed through the user's encryption context using Account-Secret-derived signing material inside the attested processing path. This gives the same verification property the system needs for release decisions: the VM can later verify that the consent record is bound to that user's cryptographic identity and to the specific sharing scope.

Confidential VMs verify the consent proof before releasing data to external services. Without a valid proof that can be verified against the user's keys and requested sharing scope, the VM's release path refuses to transmit data regardless of any other configuration or request. This prevents the approved VM code path from sharing user data without explicit, cryptographically verified authorization.

The standard cryptographic consent model uses a maximum TTL of **7 days**. Client apps renew the consent record whenever they are in active use, similar to how account keys are refreshed. If the consent expires without being renewed, Confidential VMs stop releasing data to the external service until the user's device renews the consent. Any integration that requires longer-lived authorization should be explicitly scoped, documented, and reviewed as an exception to the standard TTL model.

Users can revoke consent at any time by deleting the consent signature from the database. Once deleted, Confidential VMs can no longer verify authorization for that sharing relationship and stop releasing data to the external service. Any caches of consent state must honor revocation and the maximum TTL. Revocation prevents future releases by Bee; already transmitted data follows the external service's own retention and deletion policies and controls.

8.1 External Integrations

External integrations are controlled release points from the Confidential VM boundary. Bee uses tightly scoped Lambda proxies and provider APIs for generic external integrations. The Confidential VM release path is responsible for checking consent before sending user content, provider tokens, or action requests to those integrations.

Push notifications are treated as a special encrypted or low-content channel. For ordinary user-scoped pushes from an encrypted context, the server creates a notification-specific envelope key, encrypts the intended title and body, and sends push providers only generic display text plus `encrypted`, `keyPath`, `encryptedKey`, and `encryptedContent` fields. The mobile notification extension reconstructs the envelope from the user's locally stored account keys and replaces the generic title and body after local decryption. Delivery paths that do not have the recipient's encryption context, such as some cross-user or fallback notifications, must use minimal non-sensitive text. Richer external actions require a consent record and are constrained by the provider's own scope and retention model. Bee's enforceable guarantee is that future releases require valid consent; once content is transmitted to a third party, it follows the provider's retention and deletion model.

9 Non-Cryptographic Protections

On top of cryptographic protections, the system implements multilayer organizational controls intended to prevent any single team from accessing user data. The Bee team does not control the transparency service and cannot deploy software that would not appear in the transparency log. Conversely, the transparency team can commit entries but cannot deploy software since they do not have access to Bee's accounts directly, nor to the Private CA which is managed in a tightly controlled account accessible only by the Bee team. This separation of duties means deploying unauthorized code requires coordination across independent teams.

AWS Service Control Policies (SCPs) are enabled across all accounts, limiting what actions are possible within AWS infrastructure. Modifying these policies requires multiple levels of approvals within the organization, reducing the risk of unilateral changes to security boundaries. The Bee build pipeline is isolated from broad production access, providing an additional layer of separation from broader organizational access.

9.1 Logging, Metrics, and Diagnostics

Privacy-preserving systems still need operational telemetry: boot progress, health checks, queue depth, Redis connectivity, model-service availability, certificate verification failures, and aggregate error rates. Bee distinguishes this telemetry from user plaintext. VM boot logs, Docker logs, Prometheus metrics, and CloudWatch bootstrap events are intended to report operational state rather than decrypted content. Application code and integration proxies should log identifiers, counts, durations, queue states, request metadata, and failure classes, not raw transcripts, prompts, messages, notification bodies, provider tokens, request payloads, or decrypted records.

This control is enforced through measured software, deployment discipline, and review rather than encryption alone. Approved code running in a Confidential VM can see plaintext while processing and therefore must avoid writing plaintext to logs, metrics labels, traces, or crash reports. Diagnostic tools that run inside the VM are part of the measured manifest and subject to the same egress restrictions and transparency expectations as application code. Remote shell, SSM, EC2 Instance Connect, and ordinary interactive administration are removed from Confidential VM images, so diagnostics must be performed through measured software paths, logs, metrics, health checks, and replacement deployments.

Operational Surface	**Allowed Evidence**	**Guardrail**
Boot and certificate issuance	Boot phase, manifest identifier, PCR match result, certificate issuance status, and failure class	No interactive shell or ad hoc host inspection
Application health	Queue depth, request counts, latency, aggregate error classes, cache availability, model-service status	No raw transcripts, prompts, messages, provider tokens, decrypted records, or plaintext-bearing metric labels
Diagnostics	Measured log paths, health endpoints, structured error codes, and replacement deployments with updated measurements	No SSM, SSH, EC2 Instance Connect, or mutable debugging agents on Confidential VMs
Incident response	Stop traffic, replace instances, rotate affected keys, revoke consent, disable integrations, and deploy a new measured configuration	No live mutation of a plaintext-processing host outside the measured manifest

9.2 Operational Change Control

Bee deploys by replacing VMs with new measured images and manifests rather than mutating long-lived machines in place. The deployment pipeline builds or selects a base AMI, constructs a manifest of containers, files, S3 downloads, certificates, bridges, and secrets, registers the expected measurements with the transparency flow, and then launches new ASG instances. Secrets that are made available to containers are pinned by concrete version identifiers in the manifest, and runtime startup verifies the returned secret version before adding it to the container environment. Public exposure and unprotected endpoints require explicit high-risk configuration flags in the manifest and AMI flavor selection.

This makes operational change visible at the cryptographic identity layer. Adding a shell, changing a container image, enabling a new bridge, adding a model file, changing an egress integration, or modifying the security agent changes the measured configuration. Conforming clients and internal mTLS verifiers reject peers whose attestation and transparency proof do not match the accepted configuration. Organizational approvals and SCPs add defense in depth, but the primary control is that runtime identity changes are observable to clients and auditors.

9.3 Deployment and Change Gates

Any change that can affect plaintext handling is expected to appear as a new runtime identity. This includes base AMI changes, runtime manifests, container image digests, public exposure flags, bridge definitions, model files, S3 artifacts, bootstrap scripts, secrets made available to the workload, and security or diagnostic

agents. The security review question is therefore not only "who approved this deployment," but also "does the attested identity presented to clients correspond to the reviewed deployment."

Rollback follows the same rule. A safe rollback replaces instances with a previously measured and accepted configuration; it does not mutate a running Confidential VM back into shape. Sensitive settings, such as public reachability or unprotected exposure, are explicit high-risk configuration choices and should be reviewed as part of the manifest and AMI flavor rather than inferred from network state after launch.

10 Security Considerations

Bee Private Compute materially raises the bar for accessing user plaintext: an attacker would generally need to compromise the client, compromise approved measured code while keys are active, or obtain authorization through the user's consent path. Standard databases, caches, load balancers, operator consoles, shell access, and non-attested internal services are excluded from the intended plaintext path. AWS Nitro provides the confidential-computing foundation for infrastructure isolation and no-operator-access properties[3][5][13], while Bee's architecture adds client-held roots, attestation-gated key release, transparency-recorded runtime identity, and bounded key retention.

The remaining risks are the ones expected in any system that must process plaintext to provide a service: bugs in approved code, compromised client devices or mobile platforms, and already-authorized third-party retention. Bee mitigates these risks with periodic security audits, separation of duties between deployment and transparency functions, permanent transparency records for approved runtime configurations, source and manifest review, network isolation, narrow egress, and 7-day key expiration that bounds the active exposure window by recent client activity. Bee Private Compute was independently assessed by Trail of Bits in 2026; the full report, including the status of each finding after remediation, is published alongside this paper [14]. Bee also tracks ongoing research into the limits of attested TLS, in particular the difficulty of binding attestation to a specific instance rather than to approved software, and treats stronger instance-level binding as an open hardening area alongside the non-targetability controls in Section 4.3.

11 Ongoing Improvements

Bee Private Compute will continue to evolve as threats, product capabilities, and the underlying platform change, and Bee's threat model and mitigations will evolve with them. Bee may update this paper to document subsequent improvements to privacy, security, transparency, and resilience, and welcomes the findings from the security research community.

Acknowledgements

The authors thank Mohsin Ismail, Abhi Menon, Kunal Haryani, Chandana Reddy Madasani, Marcelo Shen, Sam Golden, and Akshay Dongala for their contributions to the systems described in this paper and for their review of drafts. We also thank J.D. Bean and Matt Wilson of AWS for their review and feedback, and Maria de Lourdes Zollo, co-founder of Bee, for her guidance and support throughout this work.

12 References

[1] Amazon EC2 Instance Attestation: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/nitrotpm-attestation.html`

[2] Isolate Data from Your Own Operators: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/isolate-data-operators.html`

[3] AWS Nitro System No Operator Access: `https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/no-aws-operator-access.html`

[4] Data Protection in Amazon EC2: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/data-protection.html`

[5] AWS Nitro System Gets Independent Affirmation of its Confidential Compute Capabilities: AWS Compute Blog, May 9 2023

[6] AWS NitroTPM: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/nitrotpm.html

[7] Validate a NitroTPM Attestation Document: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/nitrotpm-attestation-document-validate.html

[8] The Components of the AWS Nitro System: https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/the-components-of-the-nitro-system.html

[9] Model Checking Boot Code from AWS Data Centers: https://link.springer.com/chapter/10.1007/978-3-319-96142-2_28

[10] Sigstore Rekor: https://docs.sigstore.dev/logging/overview/

[11] Sigstore Security Model: https://docs.sigstore.dev/about/security/

[12] Hierarchical Deterministic Wallets: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[13] AWS Confidential Computing: https://aws.amazon.com/confidential-computing/

[14] Trail of Bits, *Amazon Bee Private Compute: Security Assessment with Fix Review*, 2026: https://github.com/trailofbits/publications/blob/master/reviews/2026-09-amazonbeeprivatecompute-securityreview.pdf