Confidential Computing Governance Framework

1 Contents

2 Abstract

This document proposes a detailed framework for *Confidential Computing* (CC) governance in the context of a regulated institution utilizing CC in the public cloud. Most considerations listed herein apply to private clouds as well, however, the intended audience for this document consists primarily of public cloud providers as well as ISVs looking to build CC offerings. Whereas a lot of current discussion around CC involves data-in-use protections, this document includes, on an equal footing, code-in-use protections as well.

3 Motivation

JPMorgan Chase & Co. is representative of a large, global, heavily regulated institution with a strong interest in strengthening protections around data processing entrusted to the public clouds. The new protections offered to code- and data-in-use are thus something we're eager to operationalize. We also know that once our many regulators wake up to the security properties of CC, it will only be a matter of time until these protections become part of the regulatory requirements we *must* abide by – much like what has happened with protections afforded to data-in-transit and data-at-rest before.

To be sure, compliance does not guarantee security, while secure solutions alone, even if well engineered, will not automatically be treated as compliant by regulators. In addition, any picture is not complete without also mentioning two other concerns of a well-governed system: resilience and cost. All these areas: security, compliance, resilience, and cost, are intertwined and affected by idiosyncrasies of CC. Ultimately, *one can only achieve a goal if one can measure and assess the current system state and monitor progress against that goal*. This is what this paper is all about.

This document sets a "north star", or the desired future end-state, and we do understand that progress towards the stated goals will be gradual and asymptotic. While this paper will argue that a lot of changes are needed to sufficiently retool cloud computing for CC adoption, the reader should not walk away feeling that wholesale changes are a *prerequisite* to adoption.

4 Background

CC offers unprecedented protections around code- and data-in-use, even when computed on by machines that the owner of the code and data in question does not administer or have in its physical possession. Unlike *Fully Homomorphic Encryption* (FHE) or *Secure Multiparty Computation*[1] (SMPC), CC executes native code in the instruction set of the processor on which it runs, compiled using commodity

[1] CC can be used for secure multiparty computation also, but the common use of the term SMPC today implies cryptographic-only (as opposed to trusted-hardware-based) solutions.

toolchains and specially packaged for execution inside *Trusted Execution Environments* (TEEs). Unlike CC, FHE and SMPC do not require either specialized hardware, or trust in hardware on which the code is executed. Both FHE and SMPC algorithms are generally less performant and less general-purpose than CC. This paper will focus on CC only.

The phrase *Confidential Computing Governance* as used in this paper refers to a set of mechanisms for ensuring that:

- Sensitive code and data are only entrusted to systems that *provably* satisfy the data owner's policies[2]
- All entities handling, or affecting secure handling of, confidential code and data can be *provably*, correctly, and timely updated to introduce new functionality, improve code quality, and/or remove security vulnerabilities
- Misbehavior can be identified in a timely fashion, correctly and unambiguously attributed to responsible parties, and *provably* and expediently corrected

Note how every bullet point above contains a reference to provability. That requirement cuts across everything governance-related and should be called out separately:

- There is comprehensive, timely and accurate reporting of all security-sensitive states and state transitions, such as which code is allowed access to what data, which security configuration settings are in place, etc.

4.1 What Makes an Application Governable?

CC is, at its core, a radical reduction of what is in the TCB of a deployed application. Aside from the way the application is engineered, trust in a deployed application – the result of making the application *governable* – must primarily be rooted in the combined trust in hardware, AS(s) and the supporting *Hardware Security Module(s)* – HSM(s). More on each below.

4.1.1 Hardware

CC is impossible without trust in hardware. The functionality provided by the hardware itself may evolve even without changing its silicon, by introduction of various microcode and firmware updates – the process that is manufacturer-specific. Hardware governance is extra challenging to a public cloud customer because it is always and entirely under the control of the cloud operator. In the world of CC, the customer has a strong interest in having visibility into the trustworthiness of the hardware, irrespective of the cloud service. Attestation is what makes such visibility possible.

4.1.2 Attestation Service

Attestation Service (AS) is one of the most fundamental roots of trust for any CC application, and trust in it must be absolute. One cannot have a confidential application where the cloud provider is in the TCB of the AS. However, AS is unlike other confidential applications and services. If it is operated by a public cloud provider, then establishing trust in the attestation service itself also requires attestation – the "bottom turtle" problem. Another possibility is operating it on-premise, and yet another – operating it

[2] As of this writing, no scenario has been considered where code and data owners are separate entities. This may change in the future.

like some highly security-sensitive customers operate HSMs in public datacenters: by having their own hardware in a physical cage with very strict ceremonies around access, provisioning, and operation.

4.1.3 Hardware Security Modules

This paper will say little about HSMs as they generally represent a well-established and well-understood component of public cloud infrastructure. CC requires interoperability between HSMs and ASs, but such interoperability can be achieved at software and policy level and does not represent a significant change to governance approaches or requirements.

4.1.4 Control Plane

As it exists today, the cloud control plane is not to be trusted, and neither is the cloud Identity and Access Management (IAM) stack – whoever can decide who has what privileges in your application, may end up gaining full control over the application. Until this fundamental lack of trust is addressed in the future, confidential applications must be engineered in a way that does not rely on any "unenlightened" aspect of the control plane, including the cloud IAM, for the confidentiality protections they offer.

4.1.5 State Capture

Governance requires an accurate, sufficiently complete, attributed, tamper-evident, timestamped, lossless, and queriable data backbone that any confidential application, service, or device can plug into.

The CC "trust stack" is illustrated in *Figure 1*. A careful reader would have noticed that the diagram is missing "supporting services" – other cloud services that the application may depend on for its capabilities. Trust in supporting services is likewise established using the AS, so it is safe to omit.

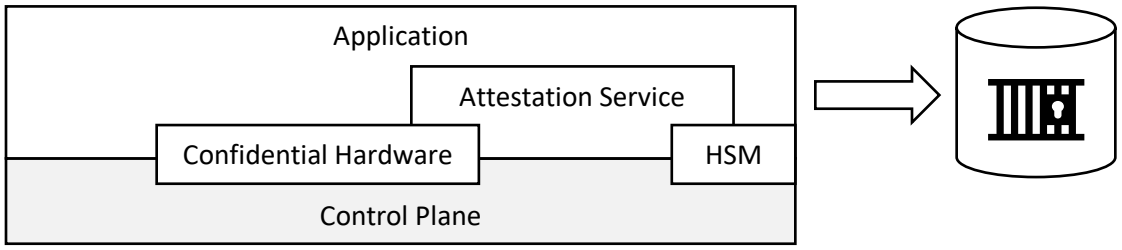

Figure 1: Confidential Computing "Trust Stack"

5 Governance Building Blocks

5.1 Outline of the Proposed Framework

This paper proposes a framework for thinking about CC governance that consists of several component parts, listed below. A separate subsection goes deeper into the discussion of each of these building blocks, followed by a full section covering governance requirements.

1. **Code** – executable logic that has direct access to, or may otherwise meaningfully influence the computation over, sensitive data; this term also encompasses all logic involved in reporting and auditing of security-sensitive events concerning such data processing
2. **Configuration** – collection of all settings that *code* uses directly in making decisions on how to act on data and how to respond to security-sensitive external stimuli, such as attempts to debug the code

3. **Policies** – sum total of all logic and settings external to the *code* that affect decision making around which *code* to execute, in which *configuration*, and what to do in security-sensitive situations; *policies* also encompass *controls*
4. **Ceremonies** – processes and protocols involving human beings, invoked when creating, evaluating, and deploying *code*, *configuration*, and *policies*, as well as dealing with reports and outcomes
5. **Enforcers** – outside parties and mechanisms that collectively ensure that the contracts underlying all previous items on this list are enforced; these include warrants, courts, governments, arbitrators, regulators, and the like

5.2 Code

Code itself comprises multiple sub-categories, discussed below.

5.2.1 Application Code

Developers write applications that are intended to run confidentially, meaning protecting code and data in use. Under the CC paradigm, data, whether operated on or generated by code, are almost always assumed to be sensitive and deserving of leak and tamper protection. Depending on scenario, the code itself might also be secret (for example, a bank might want to execute a proprietary high-frequency algorithm in the public cloud and does not want the algorithm itself to leak). The code will only be assured of running confidentially if every other supporting mechanism listed below is operating properly as well.

5.2.2 Platform Code

Platform code refers to the logic provided by the hardware manufacturers in support of CC. Depending on the nature of the device, this involves microcode and/or firmware and is often proprietary.

5.2.3 Infrastructure Code

Secure execution of code may depend on supporting infrastructure, such as identity and access management systems, ASs, key vaults, and controls. Infrastructure code is typically created by parties (such as cloud providers and ISVs) that are neither the platform manufacturer, nor the application software developer, and may (or may not) itself be running "confidentially", meaning utilizing CC hardware.

5.2.4 Supply Chain

Last but not least, even the best written code may be subverted if the toolchains involved in building it, or components it incorporates, are flawed. The entire code supply chain – libraries, compilers, code signing mechanisms and everything else involved in generation and deployment of executable code, whether application, platform, or infrastructure, fits in this category.

5.3 Configuration

Configuration settings apply to all types of code listed above, and in all cases, *bad configuration can make good code do bad things*. Configuration settings may also be specific to the supply chain components of the resulting code, such as libraries employed by the code or frameworks on which the code depends.

5.4 Policies

While configuration is "intrinsic" to the code (meaning that the code itself reads the configuration settings and acts on them), policies are "extrinsic" to it – policies govern code execution in some prescribed way, over which the code itself has no control. Policies may dictate what code runs, which configuration that code is allowed to run in, when the code might be started and terminated, at what rate the data are fed to the code, and much else.

Under the proposed CC governance framework, policies also include various controls. Controls are policy-driven infrastructure mechanisms that govern execution of code along several dimensions listed below.

Control Type	*Function*	*Examples*
Preventive	Acts to prevent undesirable outcomes from occurring in the first place	Code scanning prior to deployment Physical security barriers Logical access control mechanisms
Detective	Acts to detect deviations from desired state	Virus scanner Endpoint scanner Intrusion detection tools
Corrective	Acts to remediate an undesirable condition flagged by a detective control	Terminating a misbehaving process Rebuilding and redeploying a misbehaving or vulnerable application
Inline	Offers ongoing protection as part of an application (as opposed to being external to it)	TLS endpoint Application-level access control
Compensating	Provides defense-in-depth in cases where another protection fails or is not available	Application-level encryption to safeguard against volume-level encryption being turned off

Table 1: Types of controls

Note that defined this way, the term "policies" may include code that executes these policies. In terms of requirements, code executing policies is subject to code governance requirements, and configuration of that code is subject to configuration governance requirements.

5.5 Ceremonies

A *ceremony* is a protocol involving humans (in contrast, protocols not involving humans, such as TLS, are just code). In most information technology systems, it is humans that define and agree on policies, authorize changes, intervene in emergencies (e.g., "break glass" scenarios), and are ultimately held accountable for mistakes and misbehaviors. Governance is impossible without humans performing these tasks, and the job of machines is to make the required human tasks minimal, tractable, and effective by giving humans accurate, contextual, specific, and timely information with which to make decisions and take actions.

5.6 Enforcers

No contract is worth the medium it is stored on without corresponding enforcement mechanisms to compel parties to a contract to abide by its terms. The practices, rights and expectations of enforcers must be considered when designing the system. For instance, if an enforcer (such as a regulator) is known to be interested in audit logs generated by a system, the system must be designed to emit such

logs. If an enforcer may be expected to serve a warrant on a customer or operator of a system, ceremonies, policies, configuration, and code must be put in place to satisfy such demands.

5.7 Additional Notes

5.7.1 Impact of CC on the Shared Responsibility Model

All users of cloud services must agree on a *Shared Responsibility Model* (SRM) **[1]** with each cloud provider. This is commonly understood to mean that the cloud operator is responsible for secure operation of the cloud infrastructure itself, while the customer is responsible for the security of their environment. CC brings a set of additional considerations to the mix, as CC implies that the cloud provider removes itself from the ability to peer inside the users' sensitive code and data.

Note: there is some disagreement in the CC community on this last point. AWS has engineered their entire CC offering around a model where the customer must accept AWS being in their TCB: the customer must trust Amazon-designed hardware, its internal security processes, controls, and boundaries, as well as certain software (e.g., the Nitro hypervisor), as well as its Nitro AS. This list is not exhaustive. This document is written with the end goal, however remote, of removing the cloud provider from the customer's TCB.

A discussion is warranted with cloud providers regarding what the changes to the SRM will be made for CC. A few thoughts added below to seed the conversation:

- For cloud providers that utilize CC-capable hardware from outside silicon vendors to deliver their offerings, one can expect keeping the hardware up to date with the latest microcode and firmware fixes from the hardware vendors. For Intel, this may mean utilizing the Intel SGX Data Center Attestation Primitives (DCAP) **[2]**.
- For cloud offerings on which the security of the overall solution critically depends, e.g., AS, the customer may demand assurances (perhaps in the form of an independent audit) of the service security and operational characteristics, or, alternatively, for an option to host their own service, which means having all interfaces of the service well-documented, enabling plug-and-play replacement, or insist that the service be open-sourced with a transparent and auditable supply chain.
- The hardware and services involved in the CC end-to-end operation may require a TBD industry-wide agreement on an enhanced version of the SOC 2 **[3]** certification standard.

5.7.2 Impact on the Control Plane

Code in CC solutions cannot treat the cloud control plane as trustworthy, any more than it can treat the cloud data plane as trustworthy. To address these concerns, many parts of how the control plane operates today must be revisited with CC in mind.

For instance, any code that runs in a distributed fashion, including decentralized code, must contend with periodic software upgrades. Depending on the availability SLAs, such changes may need to be rolled out without incurring downtime, meaning running old and new code side-by-side for a period of time. The changes may be limited to just configuration (e.g., strengthening a security-sensitive configuration setting). Whereas CC solutions tend to require rollback resistance (since it shouldn't be possible to downgrade a newer version of a system to an older, potentially vulnerable one), rolling out an update is usually the period of time when rollbacks are allowed by design, since the newer version

may contain hidden flaws that only surface in production. This can be addressed via key management policy changes, and will be discussed later in this paper.

Orchestration in general is an area that will be impacted strongly by CC. From the way the nodes and workloads are examined to establish their suitability for the task, to how keys are provisioned to the workloads, will have to change to ensure end-to-end security of the application, while maintaining its reliability and performance.

A set of changes is proposed later in this paper that will have to be adopted by all orchestration frameworks in order to accommodate the new requirements placed on them by CC.

6 Governance Requirements

This section lists requirements to each of the building blocks of the proposed framework. The requirements are listed without mentioning means by which compliance with the requirements can be ascertained. Where such determination is possible, the discussion of the corresponding controls is discussed in "Requirements/Controls Matrix" at the end of this document.

6.1 Code

All code must be subject to the requirements listed in the following table.

Requirement	*Explanation*
Identifiable	Reducible to one or more unique, unpredictable, and irreversible identifiers such that any change in the code results in different, also unique, unpredictable, and irreversible, identifier value(s), as well as creates a unique binding between the code's measurement and its provenance. Typically achieved by putting the binary form of the code through a cryptographic one-way hash function. The ability to measure code is a prerequisite to being able to attest it. Note that this requirement holds even if the code being executed is itself confidential.
Circumscribed	The core expectation of data-in-use protection is that only blessed code is allowed operate on the data. In more sophisticated scenarios, the sum total of all such code may not be known at load time. If code can morph at runtime (e.g., by loading additional libraries), the entirety of all code that might be subsequently loaded must also be measured (even if not yet ready to execute) prior to attempting attestation. The order in which code is loaded post-measurement must not affect the results of code's execution[3]. The expected measurements may be expressed as a policy, as opposed to just which code hashes are valid (e.g., "all code from publisher X matching product Y with version greater than Z") and it must also be possible to deliver assurances that the policy would be subsequently enforced by the attested environment.
Patchable	It must be possible to replace code with an updated version in order to introduce new functionality, fix bugs, and/or remove vulnerabilities. Some code (e.g., an IPL loaded from ROM) may not be patchable, and a bug in such code may render the device unusable once a vulnerability is exposed or exploited. The "circumscribed code" requirement (above) dictates that patching a security defect *must* involve tearing down the defective TEE and loading new code, followed

[3] For an example of what happens when this requirement is not met, see **[4]**

Requirement	*Explanation*
	by attestation, before the patched version can regain access to data and continue execution. The flipside of this requirement is rollback protection – once updated, the code must not be allowed to revert to an older, vulnerable state, unless permitted to do so by policy.
Auditable	If a question arises whether a piece of code is defective, especially in cases where there may be reasons to suspect that a security backdoor may have been inserted, whether negligently or maliciously, a mechanism must be in place to perform an audit of the code, in order establish the root cause, assign blame, and seek redress. Auditability requirement is likely going to be subject to clearly defined ceremonies and enforcement, especially for closed-source code.
Repeatable	This is a toolchain requirement – it means that it ought to be possible for the toolchain to generate, on request, the same output (binary code) for the same set of inputs (source code, libraries and build configuration).

Table 2: Governance requirements to code

In addition to the requirements just listed, the providers of all code categories are expected to 1) issue patches when bugs and vulnerabilities are discovered, 2) make it easy for interested parties to distinguish updated versions from older ones and 3) ensure that no security backdoors are introduced by updates[4].

6.2 Configuration

Configuration, just like code, often affects the security posture of a system. As such, all configuration is subject to the following requirements:

Requirement	*Explanation*
Identifiable	It must be possible to ascertain with complete accuracy what configuration settings exist, which of these settings are security-sensitive, as well as the security implications of the corresponding configuration values, including combinations of these values. If any changes to either these settings, the expected values or their semantics are made from one release of software to the next, this information must be communicated by the creator of the software to its consumers.
Measurable	All security-sensitive configuration values must be captured in a way that allows the AS to examine them for correctness. Code and configuration are measured differently – for code, it is sufficient to obtain the hash, but for configuration – one must be able to examine the actual values of configuration settings, which one-way cryptographic functions make difficult or impossible. The configuration measurements must be securely bound to the corresponding code measurements: the AS must be able to examine them together and be sure that they correspond to the same running instance.
Immutable	It is not uncommon for code to be written in a way that picks up configuration changes at runtime. Since configuration settings are attested and resulting code issued credentials based on results of attestation, it is rarely if ever acceptable for

[4] … though for infrastructure and platform providers, only fear of discovery and ensuring reputational damage can actually prevent them from doing so

Requirement	*Explanation*
	security-sensitive configuration to change at runtime. It may fall on policy components to prevent, detect, report and/or remediate such configuration changes. Some configuration settings may be baked into the code itself, making runtime changes impossible without rebuilding and restarting code. For configuration that must be read at runtime, the corresponding code must contain safeguards against unauthorized (e.g., unsigned or insecure) configuration being picked up, as well as against unauthorized rollbacks. Code can only be securely attested after all configuration values have been ingested.
Patchable	It must be possible to strengthen the security posture of a system by changing weaker configuration setting(s) to stronger one(s). Since all configuration changes require re-attestation and subsequent reissuance of credentials to executing code, patching code is no different than patching configuration – the corresponding code most likely has to be torn down and restarted, then re-attested in the new configuration. Alternatively, a mechanism can be put in place where the control plane may deem it safe to change certain configuration setting(s) at runtime, and only then can running code pick it up. This seems like an advanced scenario, especially in a distributed system, where configuration change may result in otherwise identical nodes behaving differently until all of them pick up the change.
Auditable	This requirement, applied to configuration, is very different from the auditability requirement for code (i.e., if a question arises whether a configuration setting does what it says it does, it is still a code auditability issue). Rather, configuration auditability implies the ability to pinpoint with precision the party responsible for the current value of any configuration setting, as well as the time the value in question was last set or changed, as well as which value the setting in question has had at any point in history.

Table 3: Governance requirements to configuration

6.3 Policies

Policies involve two main building blocks: rules expressed in some kind of policy language and associated code that enforces those rules. Therefore, policy governance is a hybrid of code and configuration governance.

It helps to remember that, under the proposed framework, policies also encompass controls, and controls can be both preventive (executing prior to code being deployed) and runtime (executing alongside or even intrinsic to the code itself).

Requirement	*Explanation*
Identifiable	Since policies are often created in response to security and/or compliance requirements, it is useful for policies to have identifiers that clearly tie them to the requirements they serve to satisfy.
Measurable	Both the policy and the software used to evaluate the policy are subject to the same kinds of measurement requirements as, respectively, configuration and code; however, in this case the requirements apply to the policy text and policy engine, not the code the execution of which the policy would govern.

Requirement	*Explanation*
Auditable	All policies that are currently in place, or were in place in the past, must be attributable to a responsible party and a point in time. A separate policy may dictate who is allowed to make what kinds of policy changes. If a policy is verified prior to deployment (see below), that fact ought to also be audited.
Patchable	Like all algorithm-like constructs, policies may contain latent bugs and are likely to evolve over time. Unlike code and configuration, policies can and do change at runtime. Such changes must be reviewed, verified, approved, and audited.
Verifiable	Policies are often scriptable and resemble source code. As such, it ought to be possible to validate the policy for correctness prior to deployment. Results of such verification should be captured for purposes of auditing. Policies that execute at runtime, such as detective and corrective controls, may be subjected to targeted tests to validate their ability to perform as intended. Results of such tests ought to be audited as well.
Comprehensive	A policy should aim to cover as large a swathe of functionality in the domain it seeks to govern as practical. For instance, a code scanning tool is expected to do a comprehensive job of identifying security vulnerabilities, not just look for buffer overruns.

Table 4: Governance requirements to policies

As a general engineering rule, care must be taken to avoid an explosion in the number of different policy languages and frameworks, as that can overload humans charged with evaluating corresponding policies and enable vulnerabilities to go undetected. This is why this framework recommends picking and settling on a rich and ubiquitous policy language, and a corresponding evaluation engine (such as Rego **[5]**/OPA **[6]**).

6.4 Ceremonies

Requirement	*Explanation*
Documented	There has to exist a clear, unambiguous, and accessible explanation of each ceremony, such that each participant, as well as enforcers, have a clear understanding of the purpose, nature, and likely outcomes of each ceremony.
Vetting of Participants	This requirement applies to those ceremony participants who perform security-sensitive tasks. Vetting of participants includes a separate set of ceremonies, such as fingerprinting, background checks, issuance of credentials, etc.
Auditable	A ceremony must generate an accurate and comprehensive audit log such that it results can be examined by enforcers, if necessary.
Minimally Sufficient	More human involvement generally creates more problems (people are error-prone, may have ulterior motives, can be bribed, give different answers to the same question before and after lunch, etc.) Some ceremonies may involve “break glass” procedures for overriding the normal functioning of a system to handle emergencies. Therefore, ceremonies should only be introduced where a completely automated alternative is not feasible.
Supported by Enforcement	Any ceremony not supported by strong enforcement cannot be considered effective.
Usable	A ceremony must be designed in such a way that its users clearly understand the security implications of their actions. Failure to do so will guarantee bad decisions with reduced accountability (poor usability may result in the ability of a guilty party to deflect blame).

Table 5: Governance requirements to ceremonies

6.5 Enforcers

Little in the table below is specific to the needs of CC; it is listed for completeness and without further discussion.

Requirement	*Explanation*
Clearly Identifiable	There should be no confusion in anyone's mind regarding who (which individual(s) designated by which organization(s)) is/are acting in what enforcement role.
Trustworthy	Enforcement works because participants trust the enforcers.
Predictable	Enforcers should act in accordance with clearly documented rules. Enforcers necessarily have latitude in their decisions (otherwise they could be replaced with machines), but the extent of such latitude should be well understood by all.
Available	When the services of an enforcer are required, they should be available in clearly understood timeframes.
Powerful	Enforcers need "teeth" (their decisions are not just "advisory" but binding to all participants)
Accountable	Enforcers themselves are not absolute rulers and an appeal mechanism must be in place for dissatisfied parties.

Table 6: Governance requirements to enforcers

7 Recommendations

7.1 Confidential Computing Governance At-a-Glance

The diagram below illustrates a typical application lifecycle in relation to governance. It will be referred to throughout this section of the paper.

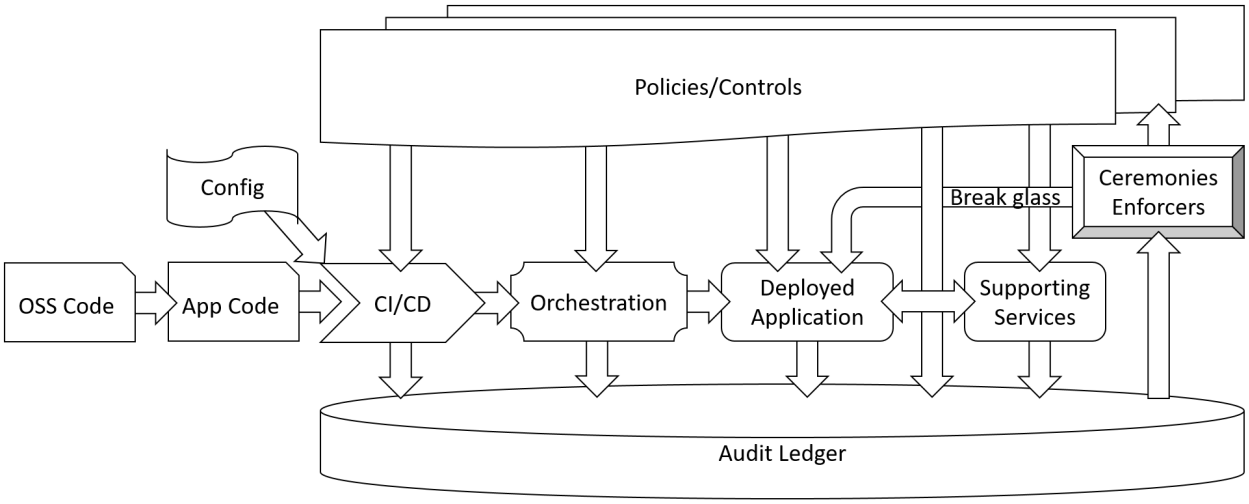

Figure 2: Confidential computing governance at-a-glance

Application development starts with a developer writing code, as well as pulling in reusable code from open-source libraries. This code, coupled with intended configuration, enters the CI/CD pipeline, the output of which is then fed into an orchestration framework which (re-)deploys the application. The deployed application executes with the help of supporting services (such as ASs, HSMs, and other middleware). The entire flow is governed by a set of policies and associated controls (e.g., if a code scan

fails, deployment would be aborted). All components of the process add evidence to an immutable ledger in the form of audit entries. The content of the ledger is used by administrators and enforcers, which, by employing a collection of ceremonies, may change the policies, as well as directly affect the running application through "break-glass" exceptions.

This is very similar to the existing IT processes, but notable differences specific to CC exist and are discussed next.

7.2 Code Enhancements

CC will necessitate rethinking of how cloud applications themselves are built, not just how they are deployed and orchestrated.

Many new patterns will emerge in support of CC, from how secrets are ingested and inputs authenticated, to how information can be securely cached while surviving live node migration, to how outputs are generated with proof of provenance. New open-source libraries will provide reusable foundations for most such common tasks. Far from being a complete list of such changes, let us include just one reusable component that would need to be built into every confidential computing application that generates output data, such that the outputs cannot be authenticated by their consumers at the moment of generation.

7.2.1 Trustworthy Output Generation

Whenever a TEE generates data, as can happen with producing regular results of computation, or when a TEE generates audit logs, it is usually important to ascertain provenance of these data, as well as ensure the data's integrity and/or confidentiality (sometimes both, never neither).

When the recipient of the generated data is available to consume it as it is sent out, the solution is simple – the generating TEE undergoes attestation, is issued a credential in the form of a certificate, a JWT or another token, and uses that to establish a secure channel with the recipient.

However, in cases where the recipient is not available to receive the data at the moment of generation, a separate set of requirement arises, since the recipient will want to know the provenance of the data. Typically, this means ensuring that the data have originated inside a proper kind of TEE that was in the best security posture possible at the time of generation.

The following table summarizes requirements to trustworthy output generation. A careful reader will have noticed that these could be neatly encapsulated in a reusable library – the subject of a separate paper, available on request.

Requirement	*Explanation*
Secure Storage	Generation of data is decoupled from its subsequent consumption; in other words, the generated data must be either or both of: • Integrity-protected (unless opted out) • Confidentiality-protected (unless opted out) Note: Safeguarding the storage medium itself is out of scope as it is no different in this case.
Non-Repudiation	The recipient of the data requires a mechanism to ensure that the generated data are authentic, meaning that:

Requirement	*Explanation*
	• The data can be verifiably attested as having originated from a trustworthy TEE, meaning that the data has originated inside the right kind (per some external policy) of TEE and its associated TCB; the "TCB" here includes any peripheral devices used in processing these data, such as forthcoming storage offload cards • The design must contend with the fact that the integrity requirements to the originating TEEs may have changed from the time when those TEE have executed to when their integrity is subsequently assessed • Authenticity of a TEE is usually provided by having it produce a "quote", meaning a signature by the platform which may include an externally generated and unguessable "freshness token" – a nonce; in the case where the eventual recipient isn't available to receive the freshly generated data, it cannot be the source of the nonce to include in the quote • The recipient may also stipulate that knowledge of decryption key must not enable the recipient to generate its own spoofed data and attribute these to the originating TEE
Resiliency	In scenarios requiring generation of a potentially large stream of data, the solution must accommodate a case where the originating TEE suffers an outage and needs to restart and resume generating data, appending the new data to the previously generated data.
Upgradability	In the case of TEE outage, the design must accommodate the possible upgrade of the TEE (e.g., the outage itself might have been caused by a buggy or outdated TEE that needed to be replaced midway by a more current one).
Compatibility	No new storage mechanisms can be assumed, i.e., the generated data must be stored in any commonly deployed storage medium, just like any encrypted data are stored today; e.g., it cannot be assumed that any additional metadata can be included alongside (but not in-band with) the generated data, unless the storage medium already supports such an option.
Performance	The solution must be able to handle generating large streams of data, or, conversely, lots of small chunks, while minimizing the latency, processing and storage overheads compared with the case where the same data are generated for immediate consumption; conversely, neither can the solution impose excessive performance penalties on the data's recipient**.**

Table 7: Requirements to trustworthy output generation

7.3 CI/CD Enhancements

Code scanning already takes place as a preventive control. For CC, a few additional considerations apply, discussed next.

7.3.1 Software Bill of Materials for Generated Images

The build process should generate a *Software Bill of Materials* (SBoM) that can be queried and referenced later, for instance, when a vulnerability is reported and an extent of compromise must quickly be determined. A number of standards are currently emerging that can help encode and query SBoMs, but it is too early to tell which one(s) will win out. The C2PA specification **[7]**, originally devised to stop deepfakes on the internet, may be helpful, but so can Software Package Data Exchange (SPDX)

[8] or CycloneDX **[9]**. The IETF is working on standardizing module identifiers **[10]** as part of its Remote ATestation procedureS (RATS) working group **[11]**. These standards, which will likely be embraced by an open-source attestation project Veraison **[12]**, can form a solid foundation for a reusable solution to standardize reasoning over SBoMs for purposes of authoring and enforcing policy.

7.3.2 Code Scanning Enhancements

Additional code scanning facilities may be employed, for both security and reliability reasons:

- Sealing data to the platform the code is running on produces a blob that cannot be unsealed if the workload is migrated to another platform instance, which may happen at any time (including when the workload is "live") in a modern datacenter. This means that sealing to the platform is something that may only be used on devices where migration is not possible (e.g., IoT gadgets), or as a means of creating a fallible cache (i.e., the code must always be prepared to deal with previously sealed data being unavailable). Such uses should be flagged by code scanning tools, and a separate API provided that handles the appropriate retry/reprovisioning logic.
- Security-sensitive configuration settings should be immutable, and changing them should require the code to be re-deployed. Language-specific code annotations for security-sensitive configuration settings may be of help here, and code scanning, as well as runtime monitoring tools, can be trained to thwart attempts to change security settings at runtime.
- Blessed security-sensitive configuration settings should be bound to the generated code images and deployed/attested together.
- Results of all security-sensitive checks that happen in the code generation phase must be logged inside the audit ledger. Similarly, all changes to policies governing preventive controls happening in the CI/CD pipeline must be audited to help identify parties responsible for any weakening of the security posture.

7.4 Orchestration Enhancements

7.4.1 Node and Workload Attestation

Two questions must be answered securely before a workload can be deployed on a node:

1. Is the node itself trustworthy?
2. Is the workload trustworthy?

A good overview of how a typical modern orchestration framework might accomplish these tasks is a write-up by SPIRE **[13]**. It is easy to see that such an orchestration framework makes all the decisions based on its own policies and configurations for the workloads entrusted to it.

The manner in which attestation of CC-enlightened nodes and workloads is performed will have to support most of these existing mechanisms, while incorporating additional CC-related enhancements. From the CC perspective, the control plane components involved in orchestration are not themselves trustworthy, any more than the networks carrying encrypted TLS traffic are trustworthy.

The node hardware in the public cloud is likely going to be administered exclusively by the cloud fabric, and only reputational risk stands in the way of the cloud operator deliberately keeping the node hardware vulnerable and/or exploiting it. That said, certain installations may give the end user a greater degree of control and options will likely exist for full hardware-level node attestation where appropriate.

A single node can host multiple workloads. These workloads will be given access to different keys, and thus different data, per policy. Most CC attestation protocols consider the node and the workload together. One use case stands apart – when code itself must also be protected against disclosure. In that case, attestation has to be done in two parts: first, the node and some "provisioning" workload are attested together, as a prerequisite to the provisioning workload being given the key to decrypt the actual "payload" code, followed by that confidential payload undergoing attestation and getting access to the sensitive data.

7.4.2 Key Provisioning

Attestation is a prerequisite to both trusting the attested workload's output and trusting it with secrets. The workload can be taught to trust the origin of the data sent to it by baking the public signing key of the sender into the workload image or its configuration. This mechanism can be used to imbue the image with other keys from the workload owner, however, the owner would be foolish to trust anything to the brand-new workload without it first passing attestation.

Therefore, a typical CC workload would start off its execution by creating CSRs to its AS, acting as a CA, around the signing and encryption keys it internally generates. These keys would then be used to secure all incoming and outgoing tunnels to other application components and the outside world, as well as to provision the workload with additional keys necessary to ingest confidential data and code.

These flows are both unavoidable and potentially expensive – something to be considered in CC application engineering. Without such care, each new compute unit launched as the application scales up would incur additional latency (as well as additional risk of downtime) related to attestation and key provisioning, even if some keys could be reused by multiple identical workloads.

Contrast this with the manner key provisioning is done today: typically, the secret keys are obtained for the workload by the orchestration framework and placed – in plaintext! – on a volume which the workload mounts, reads and then dismounts: clearly anathema to the end-to-end security of CC!

7.4.3 Upgrades and Rollbacks

CC systems understandably have an aversion to downgrades: if a system is upgraded to eliminate a security vulnerability, the supplanted version should subsequently fail attestation and no longer get access to sensitive data. This of course flies in the face of common cloud practices where a new version may need to be rolled back if it proves defective. No amount of preventive testing is guaranteed to find all such bugs: some will only surface in production.

Therefore, the orchestration framework must also orchestrate changes to attestation policies that allow for gradual introduction of new workloads and, likewise, gradual phasing out of the workloads they replace, while allowing for rollbacks to take place. Since each new CC workload must first pass attestation before it is launched, this is best accomplished by orchestrating a phasing-in and phasing-out of access to cryptographic keys.

In the spirit of comprehensive paranoia, the logic that decides which attestation and key access policies to deploy as part of such orchestration becomes the part of the orchestration framework that is itself in the TCB of the CC application, and must be engineered, operationalized, and governed accordingly.

7.4.4 Traffic Shaping and Load Balancing

Traffic shaping proxies operate at the application layer: examining HTTP headers and the like. They must either themselves be confidential and re-encrypt the traffic to the back end, or they need to switch to being network-level proxies. In a typical cloud application today, a mixture of the two approaches will be required.

Bottom line: today's solutions cannot be lift-and-shifted to a confidential model without at least a partial re-architecture, and suitable reusable components, such as confidential web application firewalls, are missing.

7.4.5 Observability vs. Confidentiality

Care must be taken such that "deep observability" which is a requirement in a lot of cloud computing world, does not end up disclosing secrets to the attackers. This can be achieved in the most straightforward manner by provisioning suitable encryption keys into every workload that generates observability data.

7.5 Supporting Services Enhancements

7.5.1 Attestation

Attestation, which is part of infrastructure supporting CC, deserves a special mention as it is an essential and frequently used part of CC TCB. Every TEE must be attested at some point after launch in order to be trusted with secrets necessary to access sensitive data, as well as to ascertain provenance of generated data (subject of the previous section). The owner of data thus has a particularly keen interest in the security, as well as governance, of AS.

Two main classes of AS must be considered, based on where the service runs:

1. AS is hosted by the owner of the data, who then takes on the burden of guaranteeing the necessary level of physical and operational security to the service and manages its policies and execution environment end-to-end
2. AS is hosted by the cloud provider, necessitating that owner trusts the cloud provider to a sufficient degree

Below we consider only the second of these options, as the first one can either be solved with today's best practices or could be folded into the second. The option where the AS is operated like a private-to-customer HSM, physically isolated inside the data center, is closer to the first option than the second.

Clearly, cloud provider maliciously influencing the decisions made by the AS is a "game over" scenario for customer's CC solutions relying on that AS, whether these solutions themselves run in that cloud or elsewhere. A provider may gain such control via several different types of attack:

- Fail to execute the AS inside TEEs and then attack the unprotected AS logic
- Run the AS inside TEEs but not keep the corresponding TEEs' platforms (their underlying hardware/software stacks) up to date, then make use of the unpatched vulnerabilities to attack the AS
- Keep the TEEs' platforms up to date, but introduce and exploit a deliberate backdoor, or utilize an existing backdoor, in the AS software

- Keep the AS software current and bug-free, but deliberately weaken the policies utilized by AS to certify customers' TEEs, and then attack those customer TEEs that are vulnerable, or certify deliberately introduced vulnerable TEEs into the customers' fabric

Note that while the AS vouches for all TEEs that are "entrusted" to its policies, and even if the AS runs inside hardware backed TEEs, the question remains of who attests the TEEs that the AS itself uses (the "bottom turtle problem").

Going through the list of governance requirements and applying them to AS, one quickly discovers that only the "Auditability" requirement of AS code is meaningfully different. The reason is that rather than rely on (recursive) attestation for AS, the AS might instead rely on a combination of auditing and code transparency. This framework strongly recommends that the AS service be open-sourced and that its toolchain is likewise precisely documented, so anyone can convince themselves that the code the AS is running is correct. The audit log of AS code generated in keeping with the requirements of Trustworthy Output Generation solution covered earlier.

7.5.2 Key Management

Key management policies must be impedance-matched to credentials issued by the AS and have provisions for phasing in and out of policies due to upgrades.

7.6 Break Glass Enhancements

If an "out of band" intervention is required, the application may be engineered specifically to allow it (e.g., by having special cryptographic keys through which it can receive appropriate commands, as cloud IAM is outside the CC trust boundary), and if it does, the audit trail must exist to fully cover the intervention in the audit log. No reusable "platform level" solution can exist here, but it is good to include such considerations here for completeness.

8 Sample Application Governance Journey

In this section, we'll walk the reader through the journey involved in turning a hypothetical high-frequency trading (HFT) application into a governable confidential computing solution.

8.1 Application Architecture

The hypothetical HFT application is illustrated below.

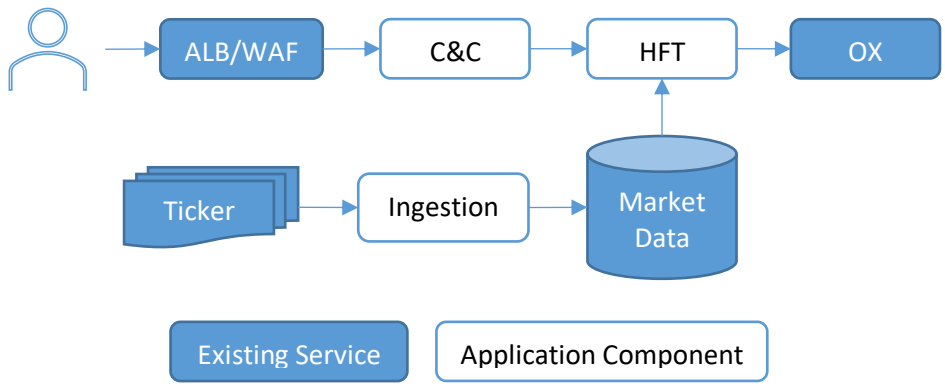

Figure 3: Hypothetical HFT application

The function of the HFT application is to translate its inputs, which consist of market data and trading strategies, into orders. The orders are generated by the HFT engine and submitted to and handled by the Order Execution ("OX") service. The user submits trading parameters to the HFT service with the help of the Command & Control ("C&C") service, which translates these inputs into trading strategies for the HFT service to follow. These command & control messages must traverse a Web Application Firewall ("WAF") and Application Load Balancer ("ALB") components before landing at the C&C service. The HFT service consumes data about up-to-date market conditions from a dedicated store, which is populated by the Ingestion service. The Ingestion service, in turn, collects market data from outside sources ("Ticker").

8.2 Assumptions & Requirements

8.2.1 Attestation Service and HSM

In the discussion that follows, we'll assume that the AS underpinning the rest of this application is:

- Already trustworthy and well-governed
- Impedance-matched to the HSM, meaning that the decisions about trustworthiness made by the AS can be honored using straightforward policy decisions by the HSM (or a CC-enabled front-end of an HSM) when deciding:
    a. Who gets access to what key material
    b. How this key material is packaged for consumption by its intended recipient

8.2.2 Security Requirements

We would like to reuse as many existing cloud services as possible, so long as such use does not endanger the critical assets of the HFT application. Although the application owners would like to utilize the public cloud for their service, they do not trust the cloud operator with any of the following:

- Trading parameters: what they are, how they change
- Trading strategies that result from these parameters
- Algorithm(s) employed by the HFT service to decide on trades based on market data and the trading parameters (but not the generated trade orders themselves – those are assumed to be public)
- Keeping market data storage medium used by the ingestion service safe from tampering, disclosure, or rollback

At the same time, certain data can be exposed, including:

- Which orders are placed and when
- Which inputs describing the outside world the application ingests
- Network/storage traffic "fingerprints" between components and the computation volume (so long as the contents themselves remain opaque)

8.2.3 Functional Requirements

In addition to the security requirements, we also have a set of functional requirements:

- High availability since HFT must operate (and scale with load) continuously during the periods when markets are open, *even if* an update needs to be pushed out mid-stream
- High performance to ensure the most profitable volume of trades

- Full attribution of all activities to responsible parties

8.2.4 Additional Assumptions

- There exists a framework that any component can use to submit audit events, and these audit events will be ordered (accurately and securely timestamped), tamper-evident, comprehensive, lossless, and trustworthy

8.2.5 Out of Scope

Break glass ceremonies must be considered for each bespoke service, and properly audited. This will not be covered here but is implicitly assumed throughout.

8.3 Discussion

Traditional cloud computing would treat the customer's computing environment in the cloud as absolutely trustworthy, but confidential computing takes a much more skeptical stance. Let's examine the differences in approaches and changes to the implementation and governance that result.

8.3.1 Ingestion Service

The job of the ingestion service is to collect market data from multiple sources and feed it to the HFT engine in a way that precludes trading errors due to incorrect, incomplete, or stale data.

The ingestion service must be able to authenticate its inputs. For that, signing is sufficient since the input data are public. The input verification keys can be either baked into the ingestion service logic, or, preferably, read from configuration. The output must be stored in a commodity storage system, but the format of the output is up to the ingestion service's inner logic, in order to imbue the output with additional security properties.

It should be possible to rotate all keys during trading hours in case of an update or key compromise, preferably without incurring downtime.

8.3.1.1 Traditional Approach

The orchestrator would procure the cryptographic key material required by the service, place these keys on a volume in plaintext and start the compute instances. During a rolling update, traffic shaping using a dedicated sidecar or traffic proxy takes place that gradually moves traffic from the old instances to the new ones.

A compute instance, when it launches, mounts the volume with the key material, reads the keys, dismounts the volume. The keys are visible to the orchestrator and thus the orchestrator is in the TCB of the service. Because of this, it is trivially easy to share keys between all auto-scaling compute instances that require them.

8.3.1.2 CC Approach

When the cloud environment is not trusted, several additional security considerations arise:

- The storage medium used for storing generated market data is subject to rollbacks and tampering
- The stored data must be protected against disclosure, and encryption-at-rest may not prevent this, especially if volume encryption is performed using cloud-provider-managed keys, as is often the case

- The secret keys must be hidden inside the compute instances, and securely procured contingent on attestation to ensure that only trustworthy information is ingested by downstream services (in this case, the HFT service)

To address these additional concerns, additional logic must be placed inside the ingestion service:

- The ingestion service must ensure that the outputs it generates are tamper-evident, meaning traceable back to the ingestion service and not some rogue source; this can be done by signing outputs with a secret key to which only the ingestion service instances have access
- Which data are selected may also be a confidential matter, but that does not *necessarily* mean that the selection logic itself is confidential – the selection criteria could be encoded as (confidential) configuration
- The ingestion service must encrypt the output data itself, instead of relying on cloud-managed storage encryption; this, likewise, is done by securely placing data encryption key(s) inside the compute instance
- The consumer of the data must be able to ascertain that the data it is operating on has not been rolled back; this can be done by the consumer service contacting the ingestion service to convince itself, or by the ingestion service securely timestamping (and chaining) all the outputs it generates

If each newly started compute instance were to create a quote, contact the AS to obtain a credential based on that quote and, with that credential, get key material from the HSM, the resulting several network roundtrips would end up making the system unacceptably fragile and slow where autoscaling is concerned. Optimizations, such as dedicated provisioning TEEs in charge of obtaining, caching and securely injecting keys into the resulting payloads, are required. This means that the TEEs hosting the ingestion service would need to be able to locally attest to these provisioning TEEs without touching the network.

It would be the job of the provisioning TEE on each compute node to attest, get the key material, properly secure it (encrypt/sign) and place the protected keys on a shared volume. Each newly created compute instance would attest locally to the provisioning TEE in order to obtain the decryption key, use that to mount and decrypt the volume on which the secret keys reside, and only then start accepting incoming requests.

Next, something must be done to ensure high-availability (i.e., no-downtime) updates. One of two possibilities should be considered: under one, the entire orchestrator framework has to be trusted (i.e., itself running confidentially and thus itself having to be governed accordingly); under another, the orchestrator is trusted only with starting compute instances and provisioning TEEs on which these compute instances depend for their key material. The latter option is the one being considered in this paper. The most sensitive operation in that process is the phasing in and out of AS policy to go from a state where only V_n of the ingestion service compute instances are allowed, to where both V_n and V_{n+1} are allowed (during the update itself, where rollback may be necessary) to V_{n+1} only, after the update was proven to be a good one. This policy change must be enforced by the provisioning TEE on each compute node, since it is the one handling local attestation for the compute instances. The component(s) tasked with modifying this provisioning the TEE local attestation policy must themselves be well governed.

Finally, the traffic shaping proxy in charge of redirecting traffic from old to new instances in case of a rolling update, is likely an L7 proxy with visibility into incoming traffic (e.g., to support sticky sessions). The ingestion service, as specified, does not have confidentiality requirements to warrant additional protections for incoming traffic since it comes from public sources in our scenario. However, it is very important to ensure is that all relevant market data are ingested, meaning that the traffic shaping proxy is not allowed to drop any incoming traffic. This requirement may mean that it has to be safe from administrative interference. Later we'll see a case where incoming traffic itself is confidential, and additional protections must be put in place.

Throughout, all security-sensitive decisions and actions must be securely audited.

8.3.2 Web Application Firewall and Load Balancing

Next, we consider how commands from the users of the HFT service flow to the C&C service. The commands are confidential as they contain trading parameters, and thus have direct impact on the trading strategies that the HFT service would execute. Confidentiality is not the sole concern: it is important that the commands affecting trading activity are actually received and acted on by the back end. Therefore, secure confirmation of receipt is also important. Finally, an adversary must not be able to affect performance or functionality by, e.g., attacking the service with invalid requests.

8.3.2.1 Traditional Approach

As is standard practice in situations where commands are sent to a cloud service, the incoming traffic must traverse a load balancer and a Web Application Firewall reverse proxy (in that or reverse order) before being routed to the C&C compute instance. More often than not, the load balancer in such cases is an Application (as opposed to Network) load balancer that terminates TLS, operates at L7, and performs additional functions, such as health checks. The WAF is a reverse proxy that handles the tasks of inspecting and sanitizing incoming data. It likewise operates at L7.

8.3.2.2 CC Approach

The L7 traffic inspection is the most troublesome aspect of WAF and ALB services, as far as CC is concerned. All cloud providers offer WAF and ALB functionalities, and the customer has the choice of either utilizing them as is, or deploying their own CC-enabled solutions. This write-up assumes the former of these two options.

There are several attack targets that the designer of a confidential service must protect when considering intervening untrusted services such as the WAF and the ALB:

1. Credentials used to authenticate the user – there can be no ambiguity about who has initiated the request
2. Target of the request (in case of HTTP, this can be expressed as the request line) – may leak information
3. Request headers (not all are security-sensitive, but some may be)
4. Request body – contains the most sensitive information
5. Timely delivery of the request to the recipient – under CC, in addition to snooping, the intervening services may act to delay transmission of sensitive information, therefore this type of attack must be detectable

Let us cover these one by one:

Authenticating the Request: the cloud control plane cannot be trusted (whoever has the power to decide who has what identity inside an application, can take it over). Therefore, care must be taken to ensure that the cloud operator cannot come in possession of the secret cryptographic keys that authenticate the request. These secret keys must be used to authenticate the body of the request. These precautions neuter many possible L7 attacks by the WAF and ALB.

Target of the Request: the request line can offer powerful clues as to what the user of the service seeks to accomplish. As such, care must be taken to divulge no sensitive information inside the request line.

Request Headers: application designers must consider request headers on a case-by-case basis and avoid putting information into the request that can expose sensitive data. Integrity of the request headers is also important, and attacks of that nature may be mitigated by adding an additional request header which contains a digest of protected headers.

Request Body: the request body must be encrypted using secrets that are known only to the submitter of the request and the recipient service. Additional key management is likely going to be required to accomplish that task.

Timely and Ordered Delivery can be ascertained by timestamping requests and responses, such that the recipient can examine these timestamps for signs of malfeasance. TEEs generally cannot be absolutely sure what time it is any moment; this is a topic of on-going research and will not be covered here at any depth. It may also be important that commands are processed in the order received, which is something the requestor and the recipient may have to coordinate, perhaps with sequence numbers.

8.3.3 Command & Control Service

The C&C service receives (confidential) trading parameters from its user(s) and computes (confidential) trading strategies which it then submits to the downstream HFT service. Many of the same considerations apply here that were already covered when discussing the Ingestion service, and will not be repeated.

8.3.3.1 Traditional Approach

Not much to add here: the functions of a front-end web service taking in parameters, utilizing them for computations and submitting the results to a back-end service are well understood. The WAF/ALB services are part of cloud infrastructure and generally trusted by the application.

8.3.3.2 CC Approach

The changes to the C&C service under the CC assumptions mirror the considerations that were discussed above when talking about WAF and ALB. The C&C service needs to implement these strategies:

- Ability to authenticate users' requests independent of the untrusted cloud IAM engine – likely using special signing keys procured from the HSM (and handed to the compute instance by the provisioning TEE, described when talking about the Ingestion service, above)
- Checking integrity of request headers using special signing verification keys that the WAF and ALB components cannot access
- Decrypting the request body utilizing decryption keys procured at startup
- Checking for dropped requests and doing other bookkeeping necessary to ensure that all inputs are properly accounted for and audited; doing this on the understanding that audit logs

themselves can be sensitive and certain information may need to be encrypted before entering the audit logs
- Establishing a secure channel with the HFT backend service (mutual TLS would work well, but keys and certificates must be obtained for that purpose)

One aspect of this list must be called out separately: maintaining a correct sequence of incoming requests may involve state management, and thus state caching. Care must be taken to recognize that any sensitive data sealed "to the platform" by the TEE may cease to be accessible if the cloud decides to migrate the compute instance to another node.

8.3.4 HFT Service

The HFT service is the most important component of the solution. In addition to all the considerations affecting other parts of the application, HFT stands out in one aspect: the code it executes is itself confidential. This means that the cloud operator is to be prevented from examining the image containing the compiled HFT code, whether prior to or during execution.

8.3.4.1 Traditional Approach

Traditional cloud computing does not have many provisions for keeping code confidential. Best it can offer is something like Virtualization Based Security in Microsoft Azure **[14]** which keeps the VM image encrypted until an authorized cloud server launches it. That approach has certain limitations – for instance, the cloud operator is not exactly out of the TCB – the hypervisor of the server on which the VM runs still can access the VM contents if hacked.

8.3.4.2 CC Approach

There are several choices available to the application designer willing and able to utilize latest CC offerings. Confidential VMs based on AMD SEV or SEV-SNP could keep the code confidential, but so can other TEE types, so long as there is a way for an encrypted image to be decrypted and launched inside another TEE.

Orchestration solutions must be updated in all these cases to allow smooth (zero-downtime) rollout of upgrades.

A reusable pattern that should work well in these scenarios is a "two-phase attest": first the compute node attests itself in order to get access to a key for decrypting and launching the sensitive code, and then attests the launched sensitive code to get access to the sensitive data. Orchestration frameworks do not currently support such two-phase attest directly.

8.4 Summary: Additional Patterns for Confidential Computing Governance

Many changes required by the components above to make full use of CC capabilities are reusable and lend themselves to be made available via OSS libraries and orchestration frameworks. Reusable and broadly applicable patterns are the key to successful adoption of this new technology.

- Provisioning TEE to speed up auto-scaling of compute instances on a single compute node
- Orchestration framework support for starting provisioning TEEs and securely (i.e., confidentially) orchestrating attestation policy changes during update rollouts
- Confidential traffic shaping proxies for secure update rollouts

- Software library for generating output from a TEE that can be subsequently (asynchronously) verified as having come from a trustworthy TEE at a later time
- State cached by a compute instance must be stored in a way that survives live migration of the compute node to another physical server, something that can happen at any time in a modern data center
- A reusable solution for TEEs to ingest encrypted code images and then decrypting and executing them, coupled with corresponding update rollout solution

9 Appendices

9.1 Terms and Abbreviations

Term	*Definition*
Attest (tr. v.)	To authenticate by signing as a witness
Govern (tr. v.)	To control, direct or strongly influence the actions and conduct of

Table 8: Terms used throughout this paper

Abbreviation	*Meaning*
ALB	Application Load Balancer
AS	Attestation Service
CA	Certificate Authority
CC	Confidential Computing
CSR	Certificate Signing Request
CoRIM	Concise Reference Integrity Manifest
FHE	Fully Homomorphic Encryption
HFT	High-Frequency Trading
HSM	Hardware Security Module
IAM	Identity and Access Management
IPL	Initial Program Loader
JWT	JSON Web Token
IPL	Initial Program Loader
ROM	Read-Only Memory
SBoM	Software Bill of Materials
TEE	Trusted Execution Environment

Table 9: Abbreviations used throughout this paper

9.2 References

1. Cloud Computing Shared Responsibility Model: https://en.wikipedia.org/w/index.php?title=Shared_responsibility_model
2. Intel SGX DCAP https://github.com/intel/SGXDataCenterAttestationPrimitives
3. SOC 2 Introduction and Overview: https://socreports.com/audit-overview/what-is-soc-2
4. "undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation": https://www.youtube.com/watch?v=rCUIJhUFA3U
5. Rego policy language: https://openpolicyagent.org/docs/latest/policy-language
6. Open Policy Agent: https://openpolicyagent.org/docs/latest
7. C2PA Specification: https://c2pa.org/public-draft/
8. SPDX: https://spdx.dev
9. CycloneDX: https://cyclonedx.org
10. Concise Reference Integrity Manifest (CoRIM): https://datatracker.ietf.org/doc/draft-birkholz-rats-corim/
11. IETF Remote Attestation Procedures (RATS) working group: https://datatracker.ietf.org/wg/rats/about/
12. Veraison Attestation Service: https://github.com/veraison
13. SPIRE: https://spiffe.io/docs/latest/spire-about/spire-concepts/

14. Microsoft Virtualization Based Security (VBS): https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs

9.3 Requirements/Controls Matrix

This section outlines which controls are possible for each category of building blocks in the proposed framework. Controls that do not have currently available implementations, which indicate potential areas for research/prototyping, are in **bold**.

9.3.1 Code Controls

	Preventive	***Detective***[5]	***Corrective***
Identifiable	Prior to deployment, the code's **SBoM is scanned** and deployment prevented if unpatched vulnerabilities are found.	After deployment, launched code cannot pass attestation (and thus gain access to sensitive data) if it contains vulnerable code. Post-deployment, **runtime asset inventory** can be referenced to identify code that has been found to be vulnerable following deployment and successful attestation.	Terminate faulty code, optionally replace with fixed version.
Circumscribed	Targeted **code scanning tool** looking for possible runtime code mutation (e.g., loading of dynamic libraries)	**Specialized tool** monitors TEEs for signs of run-time mutation.	Terminate faulty code, optionally replace with fixed version.
Patchable	Pre-release testing, audit evidence generated.	**Specialized tool** monitors TEEs to ensure no stale versions are executing and no out-of-policy code is launched.	Terminate faulty code, optionally replace with fixed version.
Auditable	**Build pipeline generates an immutable SBoM** of generated code (as well as the corresponding toolchain) and adds it, together with corresponding code measurements, to an immutable ledger.	Periodically **examine SBoM of code at run time** to validate absence of known security vulnerabilities.	Terminate faulty code, optionally replace with fixed version.
Repeatable	**Targeted testing tool** verifying reproducibility.	**Periodic audit checks of the toolchain** verifying continued reproducibility.	Raise an alarm if detective audit check fails.

Table 10: Controls for code

[5] Some of the descriptions refer to new tooling. To be effective, some of these tools would also need to run with CC guarantees.

9.3.2 Configuration Controls

	Preventive	***Detective***	***Corrective***
Identifiable	Prior to deployment, test the code and ensure it does not pick up additional configuration changes at runtime. Append test results to the build audit log.	**Specialized tool** to monitor executing TEE for signs that a configuration change not matching the code's configuration manifest is being loaded.	Terminate faulty code, optionally replace with fixed version, generate audit log.
Measurable	N/A	N/A	N/A
Immutable	Static code scanning tool ensures that no security sensitive configuration is read at runtime.	**Specialized tool** reads the generated configuration manifest and monitors the TEE for signs of updating its configuration at runtime, flagging security-sensitive misconfigurations.	Terminate faulty code, optionally replace with fixed version.
Patchable	TBD	TBD	TBD
Auditable	**Targeted code scanning tool** which looks for all instances where code reads configuration and creates a manifest, with additional smarts to discover security-sensitive configuration.	TBD	TBD

Table 11: Controls for configuration

9.3.3 Policy Controls

For now: observe that policy is to policy engine code what configuration is to regular code. This is probably an incorrect take that will not hold. Revisit this section with additional observations later.