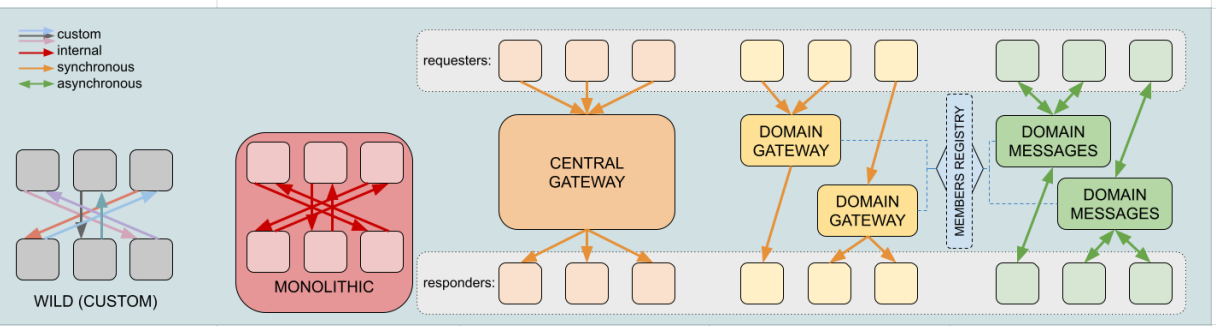
TERMINOLOGY:

whole of government architecture is NOT black and white
it consists of each different architectural styles which are expected to be interoperable with one another

DOMAIN
The subject area your software is built to handle. Domain is important for governance because the people responsible for the domain - domain experts - know the terminology, data classifications, workflows, and business rules best
If you're building software for a bank, your domain is banking.
If it's for a hospital, your domain is healthcare.
If it's for a delivery service, your domain is logistics.

SYNCHRONOUS vs ASYNCHRONOUS
synchronous request is made and waits for the response before proceeding
asynchronous request is made and handles the response later

MEMBERS REGISTRY
optional registry for authorized actors on the government data exchange ecosystem - multiple registries are possible through federation model



CORE CATEGORIES ▼	Type ▶ Category ▼	**Fragmented Point-to-Point Integration Architecture** (there is no central system nor governance nor standardization for all services, every integration is custom between the required data sources)	***Fully Centralized*** **Monolithic Architecture** (central system for all services, all services are in a single system)	***Centralized*** **Service Oriented Architecture** (services are decentralized with their own technologies, but everything uses a single central API gateway for data exchange between systems e.g. WSO2 model)	***Semi-Decentralized*** **Service Oriented Architecture** (services are decentralized with their own technologies, each has a dedicated standardized gateway using a single data exchange protocol e.g. X-Road model)	***Fully Decentralized*** **Event Driven Microservice Architecture** (services are decentralized with their own technologies, no point to point data exchange and no central gateways are used, data is exchanged with other systems using decentralized publish/subscribe message rooms, strongly supports microservices e.g. future X-Road model (X-Rooms))	RATINGS:
SYSTEM RELIABILITY AND RESILIENCE	**Single point of failure risk** (one component outage can stop the whole system)	No single core component by design, but hidden shared dependencies still exist.	One process crash stops all government services.	Central gateway can fail, government autonomous services continue working.	Gateway per service/domain limits impact, most of government services continue working.	Only dependent services may encounter conflicts, most services continue working.	GREAT
	Fault isolation blast radius (scope of impact when something breaks)	A defect can cascade through unknown point-to-point links across many systems.	A defect ripples through the whole system. Major risk.	Central gateway crash influences whole of government.	Failure stays within one service/domain.	Failures are limited to message exchange data rooms and dependent services only. Less impacted due to asynchronous exchange.	OK
	Fault isolation blast radius (scope of impact when something breaks)	Partial features can stay alive, but end-to-end service flows can fail unpredictably.	Hard to keep partial features alive when whole system is impacted.	Non-critical services can be bypassed, if need arises.	Other services continue through their own domain gateways.	Services continue working, empowered through asynchronous data exchange (data will update once systems are up).	PROBLEMATIC
	Fault isolation blast radius (scope of impact when something breaks)	Coordinating multiple systems redeploy is common.	Full redeploy is usually needed.	Restart gateway or service independently.	Restart domain gateway or service independently.	Restart individual microservice, messages buffered through message rooms.	BAD
SCALABILITY AND PERFORMANCE	**Horizontal scaling unit** (smallest deployable component you can add)	Systems can scale independently, but integration bottlenecks limit effective scaling.	Must replicate the whole stack. Often impossible, but can be done.	Services can scale independently, gateway can be scaled separately.	Scale any service and its domain gateway independently.	Any component can be scaled per-need basis.	VERY BAD
	Peak throughput capacity (maximum load that stays within targets)	Throughput is capped by the weakest dependency link and synchronous choke points.	Vertical scaling can help until hardware maxes out. System needs to be ready for peaks.	Gateway may be throttled during peaks regardless of independent system demand.	Requests are spread across multiple domain gateways, peaks handled per domain.	Everything is scaled independently, message rooms may require scaling during peaks.	
	Interaction latency (delay each request adds)	Extra network connections through multiple systems add latency.	No network hops inside the system, except internal ones (like networked databases).	Extra connections through gateway.	Extra connections through domain specific gateways.	Asynchronous connections to domain message rooms or other microservices.	
	Elastic resource allocation (ability to auto-add or shed infrastructure)	Some systems can auto-scale.	Scaling is coarse and slow.	Auto-scale gateway may be difficult, services can be scaled individually.	Separate domain gateways are easier to scale, services can be scaled individually.	Message rooms can be scaled separately and are more fault tolerant due to asynchronous connections, services can be scaled individually.	
DATA MANAGEMENT AND CONSISTENCY	**Consistency model choice** (whether data stays identical in all places at once or can differ briefly before syncing)	Consistency varies by system pair and is rarely a deliberate architecture choice.	Strong consistency due to shared databases.	Services can choose to have strong consistency or eventual consistency.	More eventual consistency is used due to separate domain gateways and services.	Eventual consistency dominates. Data will be correct eventually.	
	Distributed transaction complexity (how easy it is to make services succeed or roll back together for one business action)	True transactions across systems are almost impossible and rollbacks become business processes.	ACID (Atomic-Consistent-Isolated-Durable) and rollbacks inside one database are simple.	Gateway can broker rollbacks, but this adds overhead. Decentralized data sources increase complexity.	Multiple gateways can broker rollbacks, but this adds even more overhead. Decentralized data sources increase complexity. Rollbacks may require individual business processes.	True transactions across services are almost impossible. It is important to have separate business processes for rollbacks. Saga patterns possible.	
	Data synchronization effort (duplication and replication work)	Duplicate data is common and keeping it aligned becomes continuous work.	No duplication needed to keep in sync.	Some duplication may happen between service databases per business-need basis. Central gateway can be monitored to detect cases and resolve.	Duplicate data may exist across domains for service provision and business reasons. Authoritative master registries of data recommended for governance.	Duplicate data may exist across domains for service provision and business reasons. Authoritative master registries of data recommended for governance.	
	Audit trail visibility (ease of tracking who changed what and when)	Logs are fragmented across owners and correlation is hard to establish.	Central logs can be available for all data use. May be difficult to detect issues.	Gateway logs all traffic. Service and data exchange logs are separate and are easier to monitor/audit. Correlation ID recommended.	Domain gateways logs all traffic separately - can be sent to central target. Service and data exchange logs are separate and are easier to monitor/audit. Correlation ID recommended.	Event logs are individual and service-specific, event and message rooms can be logged separately. Easiest to detect issues. Correlation ID required for business process monitoring.	
SECURITY AND COMPLIANCE	**Attack surface size** (number of exposed entry points)	A lot of endpoints, each may be individually compromised. Compromises are impossible to detect from outside.	Fewer exposed endpoints due to data sharing happening internally within the same system. Likely easier to secure, but high risk when system security is compromised.	Central gateway is a big target.	Domain gateways are sharing similar/same software, but each are secured independently. Access less compromised.	A lot of endpoints, each may be individually compromised, but unlikely to compromise the rest of architecture.	
	Access control granularity (smallest permission scope available)	Policies differ per system and their impact is difficult to detect consistently.	Role checks often hard-coded and complex to manage and their impact to detect.	Policies are enforced centrally per endpoint. Services must still protect and validate their own access.	Policies are applied per domain and per service. Services must still protect and validate their own access.	Access policies are applied per service and per domain message room. Each component must still protect and validate access.	
	Data isolation scope (how well data is kept separate across domains or tenants)	Data separation varies by system.	Data for all domains sits together.	Services have their own data stores, gateway only handles access.	Services have their own data stores, domain gateways only handles access per domain.	Services have their own databases and are sharing data only in message rooms.	
	Regulatory audit support (built-in hooks for evidence and reporting)	Evidence gathering is manual across many systems and many log stores.	Easier to gather logs, harder to separate domains.	Central gateway logs help compliance.	Logging per domain gateway, can be sent to central source for auditing.	Event logs satisfy traceability, but many logging stores required to inspect to get the big picture.	
DEVELOPMENT AND DEPLOYMENT	**Independent deployment capability** (ship one service without touching others)	Systems can deploy alone, but dependent systems can break without warning.	Any change redeploys everything.	Services can deploy alone, gateway upgrades risk whole of architecture.	Services can deploy alone, each domain gateway can also deploy alone, only impacting its own domain and connected services.	Each service and message domain can deploy alone. Message rooms upgraded infrequently and due to asynchronous data exchange should not impact services.	
	Continuous delivery pipeline depth (steps and tooling required for release)	Many separate pipelines and integration verification is hard to automate reliably.	Simple pipeline, but releases are very risky.	Shared gateway pipeline plus per-service pipelines.	Many small pipelines.	Many tiny pipelines, team has full control.	
	Technology mix flexibility (how easily different technologies, databases etc. can run side by side)	Any technology can be used as long as it can integrate somehow.	Single language and stack. Difficult to maintain, technology migration is a huge risk.	Services free to use any technology, gateway is a fixed stack.	Services free to use any technology, gateway must use the same protocol, but may vary the stack.	Any technology can be used that can speak relevant protocols.	
	Testing scope coverage (unit, integration, end-to-end complexity)	Integration and acceptance tests are hard because services can change unpredictably and dependencies are unclear.	Unit tests are easy, integration tests are very complex.	Integration and acceptance tests are easier to set up for dependency tests, can use mocking.	Integration and acceptance tests are easier to set up for dependency tests, can use mocking.	Integration and acceptance tests are easier to set up for dependency tests, can use mocking.	
OPERATIONS AND OBSERVABILITY	**Monitoring effort required** (metrics and dashboards to keep it healthy)	Many dashboards and alert sources with limited end-to-end operational view. Some may be invisible entirely.	One set of metrics to watch.	Need gateway metrics and per-service metrics.	Need per-gateway metrics and per-service metrics.	Must watch many services separately.	
	Service discovery and configuration management load (how services find and tune themselves)	There is no consistent service discovery and configuration management.	Not needed outside.	Central gateway can expose service catalogues. Services can control over what is exposed.	Each domain gateway can expose service catalogues. Services can control over what is exposed.	Message rooms can expose the data published in the network, services can expose what functionalities they provide. A lot of governance required.	
	Runtime fault diagnosis tooling demand (need for logs, traces, profiling)	Distributed tracing and correlation IDs are needed, but rarely actually used in practice.	Technical logs intertwined, same systems.	Traces cross gateway boundary.	Distributed tracing required.	Distributed trace across events is very complex.	
	On-call operational load (alert volume and overhead for support teams)	High overhead with cross-team incident handling and slow impact assessment.	Few services, but outages are absolutely severe.	Gateway alerts can be frequent, requires time to figure out impact.	Gateway alerts are per service domains or per service, easier to trace technical issues.	Every service technical issue is individual per service or message room, quick to discover the problem.	
INTEGRATION AND INTEROPERABILITY	**API versioning strategy** (managing interface changes over time)	Interface changes break consumers and version mediation is inconsistent. Static integrations become common, slowing down business flexibility.	There is a single API for the whole system, changes are expensive and slow.	Gateway can mediate old and new versions, if dependent systems are supporting versioning.	Each gateway can mediate old and new versions, if dependent systems are supporting versioning.	Every system and message room can have dynamically evolving API versions.	
	Coupling style choice (synchronous calls versus asynchronous events)	Coupling is driven by historical integrations. There is no standardization.	Tight in-process calls only.	Mostly synchronous via gateway. Dependencies are impacted by platform management.	Mostly synchronous via domain gateways, dependencies are affected by domain management.	Coupling is completely asynchronous.	
	External integration simplicity (ease of onboarding other systems)	Outsiders must integrate multiple custom systems separately and onboarding is slow.	Single endpoint for outsiders.	Single endpoint for outsiders.	Outsiders must integrate the right domain service and the system.	Outsiders must integrate the right domain data stream service.	
	Protocol standardisation requirement (need to adhere to shared formats/protocols)	No shared format is enforced and each connection defines its own endpoint protocol.	One format is possible for data.	Gateway may translate data formats (not recommended), separate data transformation adapters possible to integrate.	Gateways may translate data formats (not recommended), separate data transformation adapters possible to integrate.	Data comes straight from the source system. Separate data transformation adapters possible to be used.	
	Infrastructure utilisation efficiency (hardware or cloud resource waste)	Duplicate platforms and uneven capacity planning can lead to waste.	Over-provision is highly likely to cover peaks.	Central gateway can become a bottleneck for services or very expensive to manage.	Domain specific gateways are scaled on-demand or configured for their specific peaks.	Idle consumers scale to zero.	
COST EFFICIENCY	**Network traffic overhead** (extra data transfer costs between services)	High volume point-to-point traffic and data duplication increase overhead.	Mostly internal memory or database calls.	One extra hop per call.	One extra hop per call, but only to a dedicated domain gateway.	High volume, but batched and brokered.	
	Tooling and licensing cost (software and platform fees)	Many different stacks accumulate significant and hidden total platform cost.	Few moving parts, big licences possible. Infrastructure is expensive.	API gateways can be pricey. Open source options exist, but infrastructure can get more expensive.	Many small gateways, open-source options exist.	Open-source options available, but due to customization options can still be pricey.	
	Staffing requirement (people needed to build and operate)	High coordination and integration maintenance drives larger staffing needs.	Smaller team, but specialists in one stack.	Gateway team plus service teams.	Primarily domain service teams.	Many small service teams.	
ORGANIZATIONAL ALIGNMENT	**Team autonomy fit** (each team owns a well-bounded service)	Teams own their systems, but custom dependencies can reduce real autonomy.	All teams touch shared code.	Teams own services, but share gateway solution.	Teams own full stack for their domain service, may share domain gateway solution.	Teams publish and consume independently.	
	Governance model style (centralised rules versus federated standards)	Governance is fragmented and standards are not enforceable across connections.	Central command works, change is slow.	Central policy via single gateway. Services are relatively independently governed.	Federation with shared protocol. Services themselves are relatively independently governed.	Central schema registry and standards for message rooms, decentralized governance for services.	
	Domain-driven design compatibility (services map cleanly to business domains)	Bounded contexts are blurred, though services can map to domains.	Bounded contexts are blurred.	Services map to domains, gateway is the single crossroad.	Services map to domains and are possible to be aligned with bounded contexts through domain gateways.	Service events and exchange maps directly to domain events.	
	Skill set demand (special knowledge staff must have)	Custom integration troubleshooting and multi-stack operational knowledge.	Fewer skills, but deep knowledge required.	Need central gateway specialists.	Need gateway protocol specialists.	Requires event modeling expertise.	
EVOLUTION AND VENDOR LOCK-IN	**Incremental migration ease** (ability to change architecture step by step)	Moving one system at a time is possible, but unknown dependencies increase risk.	Big-bang rewrites are common.	Move one service at a time behind the gateway.	Services and gateways can be adjusted per demand.	Services can be migrated per need. Rooms are more expensive to migrate.	
	Vendor lock-in risk (dependence on a single provider or tool)	Less single-platform lock-in, but many small vendor dependencies accumulate.	Tightly tied to one platform, switching is expensive.	Single central gateway creates a technology and protocol dependence.	Domain specific gateways create protocol dependence and light technology dependence.	Broker technologies can be interchangeable.	
	Cloud-native readiness (works cleanly with container orchestration and managed services)	Can be designed for containers and autoscaling, but it is not standardized.	Containers run it, but there is no fine scaling.	Gateways have cloud images, not always elastic.	Gateways have cloud images, not always elastic. Easier to scale per domain.	Designed for containers and autoscaling.	
	Experimentation speed (how quickly new ideas can be tried in production)	New functionality can ship in one system quickly, but wider rollout is slowed by custom integration dependencies.	Every A/B tweak ships the whole app.	New service implementation is easy, requires approval from central gateway.	New service implementation is easy, requires approval from domains own gateway only.	New service can be implemented requiring approval only from the dependent message rooms.	
INSTITUTIONAL COMPLIANCE AND ACCOUNTABILITY	**Privacy and Data Sovereignty** (measures ensuring data is stored, processed, and transferred according to jurisdictional, regulatory, and residency requirements)	Data flows are hard to control and even harder to prove across uncontrolled integrations.	Centralized control makes enforcing data residency simple, but increases risk if data must be segmented by jurisdiction.	Central gateway simplifies policy enforcement but creates a single concentration of data, increasing risk and limiting flexibility for cross-border requirements.	Standardized gateways allow flexible, fine-grained control over where and how data moves and is stored, supporting complex privacy needs.	High flexibility for data localization and privacy controls, but enforcement can become complex due to lack of central technical coordination.	
	Procurement and Vendor Diversity (architectural support for sourcing solutions from multiple suppliers, reducing dependency on a single vendor for contracts, operations, and maintenance)	Multi-supplier sourcing is easy, but integration coordination cost rises sharply.	Tight coupling and system-level dependencies make it difficult to mix vendors or change suppliers, increasing lock-in risk.	Central gateway is a major point of lock-in and limits flexibility in procurement and integrations.	Separate domain gateways and clear protocols enable broader competition among vendors.	Architectural openness encourages vendor diversity, but interoperability challenges can arise if standards are not enforced.	
	Transparency and Explainability (ability to provide clear, understandable access to system logic, data flows, and outcomes, supporting public oversight and accountability)	Data flows are hard to trace or explain unless strict standards are imposed.	Operations are easy to monitor and explain, but lack of modularity can obscure service logic for non-technical oversight.	Gateway centralizes oversight, making system logic more transparent and operations easier to audit, though complexity may rise with scaling.	Standardized protocols and gateways provide both technical transparency and strong audit trails for accountability.	Event-driven flows can be opaque and difficult to trace or explain for both technical and non-technical stakeholders unless strict standards are imposed.	
	Cybersecurity Risk (likelihood and impact of unauthorized access, data breaches, or system compromise due to architectural exposure and complexity)	Numerous distributed entry points increase exposure and consistent security enforcement is difficult.	Single point of access and tight coupling simplify hardening, but create a high-impact target if breached.	Central gateway consolidates control, but increases the attack surface, and a breach here affects all services.	Multiple gateways allow segmentation and isolation, reducing blast radius and enabling layered defense.	Numerous distributed components increase exposure and complexity, making consistent security enforcement challenging.	RATINGS:
	COMPATIBILITY (can this architecture integrate with the adjacent one)	✓ ▶	◀ ✓ ✓ ▶	◀ X ✓ ▶	◀ X ✓ ▶	◀ X	GREAT
	TOTAL RATING (summarized score, higher is better)	**56**	**69**	**90**	**115**	**126**	OK
		INVISIBLE IN COSTS	**LEAST EXPENSIVE ◀**	◀◀◀	◀◀◀	**◀ MOST EXPENSIVE**	PROBLEMATIC
		ABILITY TO CHANGE DEPENDS ON INDIVIDUAL INTEGRATIONS	**SLOWEST TO CHANGE AND EVOLVE ▶**	▶▶▶	▶▶▶	**▶ FASTEST TO CHANGE AND EVOLVE**	BAD
		ONLY CUSTOM COLLABORATIONS (makes things worse)	**IMPOSSIBLE FOR PUBLIC-PRIVATE COLLABORATION ▶**	▶▶▶	▶▶▶	**▶ BEST FOR PUBLIC-PRIVATE COLLABORATION**	VERY BAD

Author:
Kristo Vaher
kristo@techsterity.net