Table 1

Independence					
Inline with the “design for change” principle, a microservices architecture decomposes software into independent modules that enable graceful (less expensive) software evolution.	*An API is independently releaseable.* Any Api has to have its own runtime environment. Any API can be solely deployed to a production environment. (Docker)				
	An API independently replaceable. Any API can be changed / updated in a production environment without having to change / update other APIs.				
	An API is independently scaleable. An API can service an increasing number of requests through software scaling that does not change the scaling of other APIs.				
	An API can be built from its own unique technology stack. The provider of an API is not subject to any technical limitations that constrain using their technology stack of choice.				
	Private Data Ownership *A well-designed service doesn’t share database tables with another service.Once multiple services are directly reading and writing the same tables in a database, any changes to those tables requires coordinated deployment of all those services services. This goes against our prime directive of service independence. Sharing data storage is a sneaky source of coupling. Each service should have its own private data.* *Private data also has the advantage of letting you to select the right database technology based on the use cases of the service.*Each service needs its own *database*, possibly colocated within a shared *data server*. The key point it that the services should have no knowledge of each other's underlying database. This way, you can start out with a shared data server and separate things out in the future with just a config change.				
	If you have a single codebase for all your services, you're doing something wrong.				
	Versioning. *Versioning your API and supporting multiple versions simultaneously goes a long way to minimizing impact to other service teams. This gives them time to update their code on their own schedule. Every API should be versioned!*				
Automation					
A microservice architecture advocates for significant use of automation throughout development and software operation.	*API builds are automated.* The process used to turn source code and supporting artifacts into executable software is automated.				
	Unit testing is automated.				
	Integration testing is automated.				
	API testing is automated. The process used to validate API implementations is automated.				
	API deployments are automated. The process used to deploy the executable software for an API is automated.				
	Aim for zero-downtime updates (blue/green) If a service has to be taken down to apply an update, then every update would send little shock waves across other services. To avoid such mini disruptions (which would discourage frequent deployment), you need a way to gracefully update a service with no downtime				
Alignment					
A microservices architecture identifies and defines a set of services to build that are strategic to the enterprise. There is a clear linkage of services to their customer value proposition and to the value of building once and reusing.	*Services are built around business capabilities.* The functional domains of the business are used to organize and guide the decomposition of software into services.				
	Services are developed and delivered as products. The APIs of a service are treated as products in that (1) they are distinct offerings with a compelling value proposition for consumers, and (2) they are managed as first-class strategic corporate assets to develop and deliver.				
	API contract design is driven by consumer needs. API contract design focuses on meeting the requirements of consumer use cases, avoiding the inclusion of unnecessary functionality.				
Ownership					
Assign a custodian team that is in charge of the development, maintenance and operation of microservices that is separate from the team managing the monolithic application. This custodian team must be cross-functional, and must have full ownership of these services.	*Service team ownership is explicit and clear.* The design and implementation responsibilities for the APIs of a service are allocated to a single team at any given moment of time, affording the development organization a clear charter for completing software engineering and for clear service provider accountability to its API consumers.				
	Service ownership is consistent over time. Service ownership is sufficiently stable over time, enabling teams to become world class experts in the functional domains of the services they own, to be known service providers for these domains, and to take pride in development that leads to higher service quality.				
	Service ownership applies across the full lifecycle. Teams own services across their full lifecycle, from conception to implementation to operation to retirement and all of the phases in between. The same team does the work required for each lifecycle phase of the services it owns				
Service Architecture					
	It’s better to start with few number of coarse-grained but self-contained services. Fine-graining can happen as the implementation matures over time.				
	A service should cover a **single bounded context**. A bounded context encapsulates internal details of a domain, including any domain specific models.Ideally, you understand your product and business well enough to have identified natural service boundaries. There's no hard set rule that a service can only be one process, one virtual machine, or one container. A service consists of what it needs to autonomously implement a business capability. *Micro* in microservice has nothing to do with the physical size or lines of code, it's about minimizing complexity. A service should be *small enough* that it serves a focused purpose. At the same time, it should *big enough* that it minimizes interservice communication.				
	A well-designed service is thoughtfully stateful or stateless Instances of a stateless service don't store any information related to previous requests. An incoming request can be sent to any instance of the service. The primary benefit here is simplified operations and scaling.				
	idempotency jobs can be retried when they fail. The challenge with automatically retrying jobs is that you may not know if the failing job completed its work before it failed or not. To keep things operationally simple, you really want your jobs to be be idempotent.				
	Microservice is a single source of truth				
	The biggest challenge with shared libraries is that you have little control of when updates will get deployed across the services that use them. It could take days or weeks before other teams deploy the updated library.				
	A well-designed service has a minimal amount of database tables				
	Eventual Consistency No matter how you look at it, consistency is hard in a distributed system. Rather than fight it, a better approach to take for a distributed system is eventual consistency. In an eventually consistent system, although services may have a divergent view of the data at any point in time, they'll eventually converge to having a consistent view.				
	Is the system capable of performing Operations asynchronously. As you embrace eventual consistency, you'll find that not everything needs to be done while the request is blocked waiting for a response. Anything that can wait (and is resource or time intensive) should be passed as jobs to asynchronous workers.				
	Feature flags A feature flag is code that lets you turn on or off specific features at run time, effectively decoupling code deployment from feature deployment. This enables you to *deploy* the code for a feature incrementally over a period of time. Then, you can *release* the feature to the users when you're ready.				
	API Gateway				
	Compensatable Transaction				
Resilient					
	Can the service perform retries in case of unexpected failures?				
	Does the system have a proper/defined failure fallback?				
	Does it ensure that the application still keeps on running in the unlikely event of one of the parts going down and is it capable of preventing cascading impact of the entire system (Bulkhead)				
	Does the service has support for time-out(s)?				
	Do they have capability to fail fast and avoid any additional overhead on the system as a result of indefinite time-outs/retries. (Circuit-Breaker)				
Operations					
	API failures are detected. The operational infrastructure detects API failures.				
	API failure recovery is automated. The operational infrastructure can automatically respond to an API failure, fixing or removing problems, returning API execution back to a nominal state.				
	API execution logging is robust. A centralized logging system should be provided to the service teams by the platform. All services should ship their logs to the same logging system in a standardized log format. This approach provides the service teams with the most flexibility - ability to search across all services, within a specific service, or within an instance of a service. All from the same place.				
	API execution monitoring is robust. *Centralized monitoring should be a core component of your platfrom.* There are sufficient visualizations and/or other forms of monitoring that provide operational insight into API execution. Your monitoring solution should have the ability to aggregate metrics across instances. Additionally, you need to be able to quickly drilldown on those aggregated metrics to see their components in detail. All of this helps quickly assess if an identified failure is occurring service wide or is isolated to a specific instance of a service.				
	How will it be configured? *A centralized configuration system should be provided to the service teams by the platform. A deployment package that's deployable anywhere shouldn't contain environment specific options or secrets. The teams need the ability to manage the configuration and securely get them to the services on startup.*				
	How will it be discovered? *In an environment where service instances come and go, hard coding IP addresses isn't going to work. You will need a discovery mechanism that services can use to find each other. This means having a source of truth for what services are available. You'll also need some way to utilize that source of truth to discover and balance communication to the services instances.*				
	Correlation IDs *A single user request can result in activity occurring across many services, which makes things difficult when trying to debug the impact of a specific request. One way to make things simpler is to include a correlation ID in service requests. A correlation ID is a unique identifier for the originating request that is passed by each service to any downstream requests.*				

Security					
	Centralize Authentication				
	Centralize Authorization				
	Centralize Audit				
	Does the service has support for time-out(s)?				