**Appendix A. Monoliths vs. microservices**

This appendix evaluates monoliths vs. microservices. The author’s personal experience is that it seems many sources describe the advantages of microservice over monolith architecture but do not discuss the tradeoffs, so we will discuss them here. We use the terms “service” and “microservice” interchangeably.

*Microservice architecture* is about building a software system as a collection of loosely-coupled and independently developed, deployed, and scaled services. *Monoliths* are designed, developed, and deployed as a single unit.

**A.1 Advantages of monoliths**

Table A.1 discusses the advantages of monoliths over services.

Table A.1 Advantages of monoliths over services

| Monolith | Service |
| --- | --- |
| Faster and easier to develop at first because it is a single application. | Developers need to handle serialization and deserialization in every service, and handle requests and responses between the services.  Before we begin development, we first need to decide where the boundaries between the services should be, and our chosen boundaries may turn out to be wrong. Redeveloping services to change their boundaries is usually impractical. |
| A single database means it uses less storage, but this comes with tradeoffs. | Each service should have its own database, so there may be duplication of data and overall greater storage requirements. |
| With a single database and fewer data storage locations in general, it may be easier to comply with data privacy regulations. | Data is scattered in many locations, which makes it more difficult to ensure that data privacy regulations are complied with throughout the organization. |
| Debugging may be easier. A developer can use breakpoints to view the function call stack at any line of code and understand all logic that is happening at that line. | Distributed tracing tools like Jaegar or Zipkin are used to understand request fan-out, but they do not provide many details, such as the function call stack of the services involved in the request. Debugging across services is generally harder than in a monolith or individual service. |
| Related to the previous point, being able to easily view all the code in a single location and trace function calls may make the application/system as a whole generally easier to understand than in a service architecture. | A service’s API is presented as a black box. While not having to understand an API’s details may make it easier to use, it may become difficult to understand many of the fine details of the system. |
| Cheaper to operate and better performance. All processing occurs within the memory of a single host, so there are no data transfers between hosts, which are much slower and more expensive. | A system of services that transfer large amounts of data between each other can incur very high costs from the data transfers between hosts and data centers. Refer to <https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90> for a discussion on how an Amazon Prime Video reduced the infrastructure costs of a system by 90% by merging most (but not all) of their services in a distributed microservices architecture into a monolith. |

**A.2 Disadvantages of monoliths**

Monoliths have the following disadvantages compared to microservices:

* Most capabilities cannot have their own lifecycles, so it is hard to practice Agile methodologies.
* Need to redeploy the entire application to apply any changes.
* Large bundle size. High resource requirements. Long startup time.
* Must be scaled as a single application.
* A bug or instability in any part of a monolith can cause failures in production.
* Must be developed with a single language, so it cannot take advantage of the capabilities offered by other languages and their frameworks in addressing requirements of various use cases.

**A.3 Advantages of services**

The advantages of services over monoliths include the following:

1. Agile and rapid development and scaling of product requirements/business functionalities.
2. Modularity and replaceability.
3. Failure isolation and fault-tolerance.
4. More well-defined ownership and organizational structure.

**A.3.1 Agile and rapid development and scaling of product requirements and business functionalities**

Designing, implementing, and deploying software to satisfy product requirements is slower with a monolith than a service because the monolith has a much bigger codebase and more tightly coupled dependencies.

When we develop a service, we can focus on a small set of related functionalities and the service’s interface to its users. Services communicate via network calls through the service interfaces. In other words, services communicate via their defined APIs over industry-standard protocols such as HTTP, gRPC, and GraphQL. Services have obvious boundaries in the form of their APIs, while monoliths do not. In a monolith, it is far more common for any particular piece of code to have numerous dependencies scattered throughout the codebase, and we may have to consider the entire system when developing in a monolith.

With cloud-based container native-infrastructure, a service can be developed and deployed much quicker than comparable features in a monolith. A service that provides a well-defined and related set of capabilities may be CPU-intensive or memory-intensive, and we can select the optimal hardware for it, cost-efficiently scaling it up or down as required. A monolith that provides many capabilities cannot be scaled in a manner to optimize for any individual capability.

Changes to individual services are deployed independently of other services. Compared to a monolith, a service has a smaller bundle size, lower resource requirements, and faster startup time.

**A.3.2 Modularity and replaceability**

The independent nature of services makes them modular and easier to replace. We can implement another service with the same interface and swap out the existing service with the new one. In a monolith, other developers may be changing code and interfaces at the same time as us, and it is more difficult to coordinate such development vs. in a service.

We can choose technologies that best suit the service’s requirements (e.g., a specific programming language for a frontend, backend, mobile, or analytics service).

**A.3.3 Failure isolation and fault-tolerance**

Unlike a monolith, a microservices architecture does not have a single point of failure. Each service can be separately monitored, so any failure can be immediately narrowed down to a specific service. In a monolith, a single runtime error may crash the host, affecting all other functionalities. A service that adopts good practices for fault-tolerance can adapt to high latency and unavailability of the other services that it is dependent on. Such best practices are discussed in section 3.3, including caching other services’ responses or exponential backoff and retry. The service may also return a sensible error response instead of crashing.

Certain services are more important than others. For example, they may have a more direct effect on revenue or are more visible to users. Having separate services allows us to categorize them by importance and allocate development and operations resources accordingly.

**A.3.4 Ownership and organizational structure**

With their well-defined boundaries, mapping the ownership of services to teams is straightforward compared to monoliths. This allows concentration of expertise and domain knowledge; that is, a team that owns a particular service can develop a strong understanding of it and expertise in developing it. The flip side is that developers are less likely to understand other services and possess less understanding and ownership of the overall system, while a monolith may force developers to understand more of the system beyond the specific components that they are responsible to develop and maintain. For example, if a developer requires some changes in another service, they may request the relevant team to implement those changes rather than doing so themselves, so development time and communication overhead are higher. Having those changes done by developers familiar with the service may take less time and have a lower risk of bugs or technical debt.

The nature of services with their well-defined boundaries also allows various service architectural styles to provide API definition techniques, including OpenAPI for REST, protocol buffers for gRPC, and Schema Definition Language (SDL) for GraphQL.

**A.4 Disadvantages of services**

The disadvantages of services compared to monoliths include duplicate components and the development and maintenance costs of additional components.

**A.4.1 Duplicate components**

Each service must implement inter-service communication and security, which is mostly duplicate effort across services. A system is as strong as its weakest point, and the large number of services exposes a large surface area that must be secured, compared to a monolith.

Developers in different teams who are developing duplicate components may also duplicate mistakes and the efforts needed to discover and fix these mistakes, which is development and maintenance waste. This duplication of effort and waste of time also extends to users and operations staff of the duplicate services who run into the bugs caused by these mistakes, and expend duplicate effort into troubleshooting and communicating with developers.

Services should not share databases, or they will no longer be independent. For example, a change to a database schema to suit one service will break other services. Not sharing databases may cause duplicate data and lead to an overall higher amount and cost of storage in the system. This may also make it more complex and costly to comply with data privacy regulations.

**A.4.2 Development and maintenance costs of additional components**

To navigate and understand the large variety of services in our organization, we will need a service registry and possibly additional services for service discovery.

A monolithic application has a single deployment lifecycle. A microservice application has numerous deployments to manage, so CI/CD is a necessity. This includes infrastructure like containers (Docker), container registry, container orchestration (Kubernetes, Docker Swarm, Mesos), CI tools such as Jenkins, and CD tools, which may support deployment patterns like blue/green deployment, canary, and A/B testing.

When a service receives a request, it may make requests to downstream services in the process of handling this request, which in turn may make requests to further downstream services. This is illustrated in figure A.1. A single request to Netflix’s homepage causes a request to fan out to numerous downstream services. Each such request adds networking latency. A service’s endpoint may have a one-second P99 SLA, but if multiple endpoints are dependencies of each other (e.g., service A calls service B, which calls service C, and so on), the original requester may experience high latency.

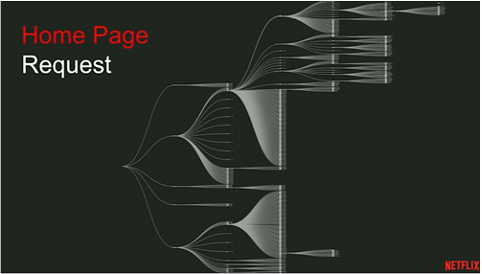


Figure A.1 Illustration of request fan-out to downstream services that occurs on a request to get Netflix’s homepage. Image from <https://www.oreilly.com/content/application-caching-at-netflix-the-hidden-microservice/>.

Caching is one way to mitigate this, but it introduces complexity, such as having to consider cache expiry and cache refresh policies to avoid stale data, and the overhead of developing and maintaining a distributed cache service.

A service may need the additional complexity and development and maintenance costs of implementing exponential backoff and retry (discussed in section 3.3.4) to handle outages of other services that it makes requests to.

Another complex additional component required by microservices architecture is distributed tracing, which is used for monitoring and troubleshooting microservices-based distributed systems. Jaeger and Zipkin are popular distributed tracing solutions.

Installing/updating a library on a monolith involves updating a single instance of that library on the monolith. With services, installing/updating a library that is used in multiple services will involve installing/updating it across all these services. If an update has breaking changes, each service’s developers manually update their libraries and update broken code or configurations caused by backward incompatibility. Next, they must deploy these updates using their CI/CD (continuous integration/continuous deployment) tools, possibly to several environments one at a time before finally deploying to the production environment. They must monitor these deployments. Along the way in development and deployment, they must troubleshoot any unforeseen problems. This may come down to copying and pasting error messages to search for solutions on Google or the company’s internal chat application like Slack or Microsoft Teams. If a deployment fails, the developer must troubleshoot and then retry the deployment and wait for it again to succeed or fail. Developers must handle complex scenarios (e.g., persistent failures on a particular host) All of this is considerable developer overhead. Moreover, this duplication of logic and libraries may also add up to a non-trivial amount of additional storage.

**A.4.3 Distributed transactions**

Services have separate databases, so we may need distributed transactions for consistency across these databases, unlike a monolith with a single relational database that can make transactions against that database. Having to implement distributed transactions is yet another source of cost, complexity, latency, and possible errors and failures. Chapter 5 discussed distributed transactions.

**A.4.4 Referential integrity**

*Referential integrity* refers to the accuracy and consistency of data within a relationship. If a value of one attribute in a relation references a value of another attribute and then the referenced value must exist.

Referential integrity in a monolith’s single database can be easily implemented using foreign keys. Values in a foreign key column must either be present in the primary key that is referenced by the foreign key, or they must be null (<https://www.interfacett.com/blogs/referential-integrity-options-cascade-set-null-and-set-default>). Referential integrity is more complicated if the databases are distributed across services. For referential integrity in a distributed system, a write request that involves multiple services must succeed in every service or fail/abort/rollback in every service. The write process must include steps such as retries or rollbacks/compensating transactions. Refer to chapter 5 for more discussion of distributed transactions. We may also need a periodic audit across the services to verify referential integrity.

**A.4.5 Coordinating feature development and deployments that span multiple services**

If a new feature spans multiple services, development and deployment need to be coordinated between them. For example, one API service may be dependent on others. In another example, the developer team of a Rust Rocket (<https://rocket.rs/>) RESTful API service may need to develop new API endpoints to be used by a React UI service, which is developed by a separate team of UI developers. Let’s discuss the latter example.

In theory, feature development can proceed in parallel on both services. The API team need only provide the specification of the new API endpoints. The UI team can develop the new React components and associated node.js or Express server code. Since the API team has not yet provided a test environment that returns actual data, the server code or mock or stub responses from the new API endpoints and use them for development. This approach is also useful for authoring unit tests in the UI code, including spy tests (refer to <https://jestjs.io/docs/mock-function-api> for more information).

Teams can also use feature flags to selectively expose incomplete features to development and staging environments, while hiding them from the production environment. This allows other developers and stakeholders who rely on these new features to view and discuss the work in progress.

In practice, the situation can be much more complicated. It can be difficult to understand the intricacies of a new set of API endpoints, even by developers and UX designers with considerable experience in working with that API. Subtle problems can be discovered by both the API developers and UI developers during the development of their respective services, the API may need to change, and both teams must discuss a solution and possibly waste some work that was already done:

* The data model may be unsuitable for the UX. For example, if we develop a version control feature for templates of a notifications system (refer to section 9.5), the UX designer may design the version control UX to consider individual templates. However, a template may actually consist of subcomponents that are versioned separately. This confusion may not be discovered until both UI and API development are in progress.
* During development, the API team may discover that the new API endpoints require inefficient database queries, such as overly large SELECT queries or JOIN operations between large tables.
* For REST or RPC APIs (i.e., not GraphQL), users may need to make multiple API requests and then do complex post-processing operations on the responses before the data can be returned to the requester or displayed on the UI. Or the provided API may fetch much more data than required by the UI, which causes unnecessary latency. For APIs that are developed internally, the UI team may wish to request some API redesign and rework for less complex and more efficient API requests.

**A.4.6 Interfaces**

Services can be written in different languages and communicate with each other via a text or binary protocol. In the case of text protocols like JSON or XML, these strings need to be translated to and from objects. There is additional code required for validation and error and exception handling for missing fields. To allow graceful degradation, our service may need to process objects with missing fields. To handle the case of our dependent services returning such data, we may need to implement backup steps such as caching data from dependent services and returning this old data, or perhaps also return data with missing fields ourselves. This may cause implementation to differ from documentation.

**A.5 References**

This appendix uses material from the book *Microservices for the Enterprise: Designing, Developing, and Deploying* by Kasun Indrasiri and Prabath Siriwardena (2018, Apress).