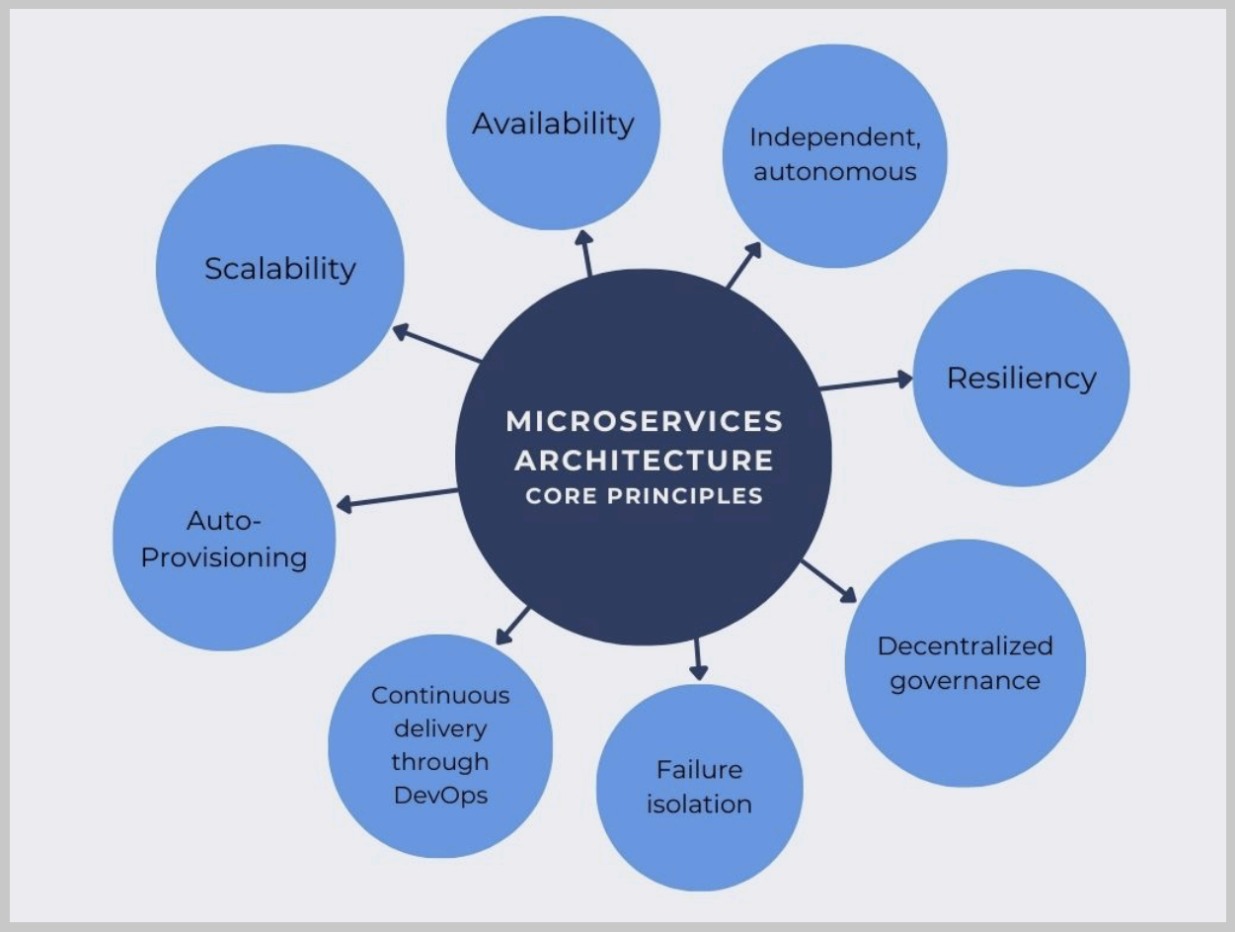
TO-DO:

https://blog.payara.fish/benefits-of-microservices-architecture

https://www.spiceworks.com/tech/devops/articles/what-are-microservices/

What makes a microservice optimized:

- Architecture & benefits
- How to optimize
- Benefits
- https://blog.payara.fish/benefits-of-microservices-architecture
- https://www.spiceworks.com/tech/devops/articles/what-are-microservices/



1) Scalability: Since each microservice operates independently, scaling individual components of an application becomes more straightforward.
2) Flexibility/Program language and technology agnostic/Technology diversity: Different microservices can be developed using different technologies, enabling teams to choose the best tools for each task.

a) Microservices architectures are language agnostic and also allows the developers to use their existing skill sets to maximum advantage – no need to learn a new programming language just get the work done.
b) Using cloud-based microservices gives developers another advantage, as they can access an application from any internet-connected device, regardless of its platform.
c) Microservices allow for the use of different technology stacks for different services. This means that teams can use the best tool for the job, which can result in higher productivity and better quality code.

3) Resilience: If one microservice fails, it doesn't bring down the entire system, thus enhancing the overall system's resilience.
4) Fault tolerance: In a monolithic application, a failure in one component can impact the entire application. With microservices, all services are independent of each other and can continue to function even if another service fails. This results in a more resilient system and reduces the risk of downtime.
5) Easy maintenance/simpler to deploy: Microservices are smaller and more focused, which makes them easier to understand, test, and maintain. Changes to one service can be made without affecting the overall system, which reduces the risk of introducing bugs or breaking functionality.
6) Agility/Autonomy: Microservices enable continuous delivery and deployment, which means that changes can be made and deployed quickly without causing other problems across the system. This allows teams to respond to changing business needs and market conditions more quickly.
    a) **Faster time-to-market:** Developers can plug this new "microsurgery" into the architecture without fear of conflicts with other code or of creating service outages that ripple across the website. Development teams working on different microservices don't have to wait for each other to finish. Companies can develop and deploy new features quickly and upgrade older components as new technologies allow them to evolve.
7) Cost-effective: Microservices can be deployed in a containerized environment, which allows for efficient resource utilization and reduces infrastructure costs. This can result in significant cost savings, especially for large-scale applications.
8) Reusability: Some microservice applications may be shareable across a business. If a site has several different areas, each with a login or payment option, the same microservice application can be used in each instance.

a) *Single Responsibility Principle*
b) Different data storage based on needs: Having a separate data store for each microservice ensures that every service is responsible for storing its own data, which reduces dependencies and tight coupling between microservices. This approach, known as data sovereignty, makes it easier to manage and scale your architecture. Each microservice should have a dedicated database, with an API in place to allow communication between the databases and the respective services.
    i) Relational Databases: Ideal for services requiring ACID transactions.
    ii) NoSQL Databases: Useful for services dealing with large volumes of unstructured data.
    iii) In-Memory Databases: Excellent for caching and fast data retrieval.
c) Asynchronous communication: Message Queues: RabbitMQ, Apache Kafka, Publish-Subscribe Systems: Amazon SNS, Google Pub/Sub
d) Containerization: bundling a microservice along with its dependencies into a single package
    i) Benefits of Using Docker for Microservices
        (1) Consistency: Ensures that a microservice runs the same way across development, testing, and production environments.

(2) Isolation: Keeps each microservice isolated, reducing the risk of conflicts and simplifying deployment.
(3) Scalability: Makes it easier to scale microservices by replicating containers across multiple instances.

e) Kubernetes: Orchestration involves managing, scaling, and deploying containers in a way that optimizes resource use and ensures reliability.
   i) automates the deployment, scaling, and operations of containers, making it easier to manage microservices at scale.
   ii) load balancing, automatic scaling, and self-healing

f) separate the build and deploy processes
   i) Build: produce a deployable artifact (Docker image) to deploy across environments without modification
   ii) This separation ensures consistency and reduces the chances of deployment issues caused by environment-specific configurations
   iii) Deployable: Tools like Jenkins, GitLab CI/CD, and CircleCI can be used to automate the build process, creating artifacts that can be reliably deployed in staging, testing, and production environments.

g) Domain-Driven Design (DDD) defining the boundaries of each microservice, ensuring that each service aligns with a specific business capability

h) Stateless microservices: necessary states should be stored in a database or an external data store. Benefit: easier to scale because don't worry about replicating session data across multiple instances.
   i) When state management is unavoidable: distributed caching solutions (Redis or Memcached) to store session data, ensuring it's accessible across different instances of a microservice

i) Micro frontends (sounds like literally everything talked about but now for frontend)
   i) Isolation: Each micro frontend should be isolated to avoid conflicts.
   ii) Independent Deployment: Allow each micro frontend to be deployed independently, reducing the risk of system-wide failures.
   iii) Consistent User Experience: Ensure a seamless user experience across different micro frontends.

j) Monitoring and observability:
   i) Effective monitoring allows you to detect and diagnose issues quickly, ensuring the system remains reliable and performant.
      (1) Tools and Techniques for Effective Monitoring:
         (a) *Prometheus*: A popular open-source monitoring solution,
         (b) *Grafana*: A visualisation tool that works well with Prometheus.
         (c) ELK Stack (Elastic search, Logstash, Kibana): For centralised logging and monitoring.
   ii) Implementing observability involves collecting and analysing data from logs, metrics, and traces to gain insights into the system's performance and behaviour.

k) Security is a significant concern in microservices architectures due to the increased number of services and their interactions. Each microservice needs to be secured individually
   i) Best Practices for Securing Microservices
      (1) API Gateway: Use an API Gateway to manage authentication, rate limiting, and security policies.
      (2) Token-Based Authentication: Implement OAuth2 or JWT for secure token-based authentication.
      (3) Encryption: Ensure data in transit is encrypted using TLS/SSL.

ii) Automated Testing for Microservices: allow quick identifying issues and ensure that your microservices work correctly before they reach production.
    (1) Types of Tests to Implement
        (a) Unit Tests: Test individual components of a microservice.
        (b) Integration Tests: Ensure that services work together correctly.
        (c) End-to-End Tests: Simulate user-interaction test whole system.
    (2) Tools for Automating Microservices Testing
        (a) JUnit: For unit testing in Java-based microservices.
        (b) Postman: For testing APIs.
        (c) Selenium: For end-to-end testing of web applications.

iii) Versioning Microservices: where multiple versions of a service may need to coexist. Proper strategies help manage backward compatibility and ensure that updates don't break the system.
    (1) Strategies for Versioning Microservices
        (a) URI Versioning: Include the version in the API endpoint (e.g., /api/v1/).
        (b) Header Versioning: Specify the version in the HTTP headers.
        (c) Content Negotiation: Allow clients to request specific versions through content negotiation.

l) Cost-effective: Microservices can be deployed in a containerized environment, which allows for efficient resource utilization and reduces infrastructure costs. This can result in significant cost savings, especially for large-scale applications.

m) Documentation:
    i) Best Practices for Creating Useful Documentation
        (1) Keep It Updated: Regularly update documentation to reflect changes in the codebase.
        (2) Use Clear and Concise Language: Make it easy to understand.
        (3) Include Examples: Provide examples and use cases to illustrate concepts.
        (4) Tools and Formats for Documentation
        (5) Swagger/OpenAPI: For API documentation.
        (6) Markdown: For writing clear and readable documentation.
        (7) ReadTheDocs: For hosting and organizing documentation.

n) 2. Proxy Your Microservice Requests Through an API Gateway
    i) An API gateway acts as a single entry point for all client requests, effectively abstracting the complexity of the underlying microservices. It is responsible for routing requests to the appropriate microservices, aggregating responses, and handling cross-cutting concerns such as authentication, rate limiting, and monitoring. By proxying requests through an API gateway, you can improve security, manageability, and maintainability of your microservices architecture.

o) Ensure API Changes Are Backwards Compatible
    i) Backward compatibility is essential to prevent breaking changes when deploying updates to your microservices. By adhering to this principle, you can ensure that existing clients can continue to use your services without any issues. When making changes to your API, consider adding new features using methods such as introducing new endpoints or adding optional parameters. If breaking changes are unavoidable, provide a reasonable migration path for clients to follow.

p) Version Your Microservices for Breaking Changes
    i) When breaking changes are inevitable, it's crucial to version your microservices appropriately. By doing so, you can allow clients to choose which version of the service they wish to use, providing them with a smooth transition. You can use various strategies for versioning, such as including the version in the API URL or

using custom HTTP headers. Ensure that you communicate these changes effectively to your clients and give them ample time to migrate to the new version.

*****TODO: Cara & Madison should go through and decide how they want to rank the info above like architecture and whatnot, Cara discuss with Madison after lunch :) ******

Steps Program can take to identify how optimal:

1) Analyze the document and break it up into a B-tree, saving the name, path, and type of document ("file", "dir", "repo", etc)
2) Define the objectives (point) of the program (search through readme?)
3) **** Based on definition, define most optimal architecture for the microservice
4) Based on the architecture identify closest architecture pattern being used, identify anti-patterns and why they negative
5) Compare these two results and decide from there which is best
6) Go through and rank for efficiency of architecture
7) Create an overall score of the microservice architecture, defining where faults lie and offer suggestions for how to become closer to "the most optimal" microservice system

REMINDER: (mainly for cara) POINT OF THIS PROGRAM IS TO TURN EVERYTHING INTO MICROSERVICE, this means that monolithic architectures can be analyzed, and program will suggest how to turn into microservice (→ don't need microservices to begin with)

- Yes, madison agrees

How we want to define the optimization of the architecture:

1) Containerization → cannot be based purely on the amount (because efficiency is relative to size of the program, we should find a way to define
2)

CHALLENGES THAT COME WITH MICROSERVICES (that we'd like to minimize when suggesting): https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

- **Complexity**. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing**. Writing a small service that relies on other dependent services requires a different approach than writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- **Lack of governance**. The decentralized approach to building microservices has advantages, but it can also lead to problems. You might end up with so many different languages and frameworks that the application becomes hard to maintain. It might be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.
- **Network congestion and latency**. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load leveling.

- **Data integrity**. Each microservice responsible for its own data persistence. As a result, data consistency across multiple services can be a challenge. Different services persist data at different times, using different technology, and with potentially different levels of success. When more than one microservices is involved in persisting new or changed date, it's unlikely that the complete data change could be considered an ACID transaction. Instead, the technique is more aligned to BASE (Basically Available, Soft state, and Eventually consistent). Embrace eventual consistency where possible.
- **Management**. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
- **Versioning**. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
- **Skill set**. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.