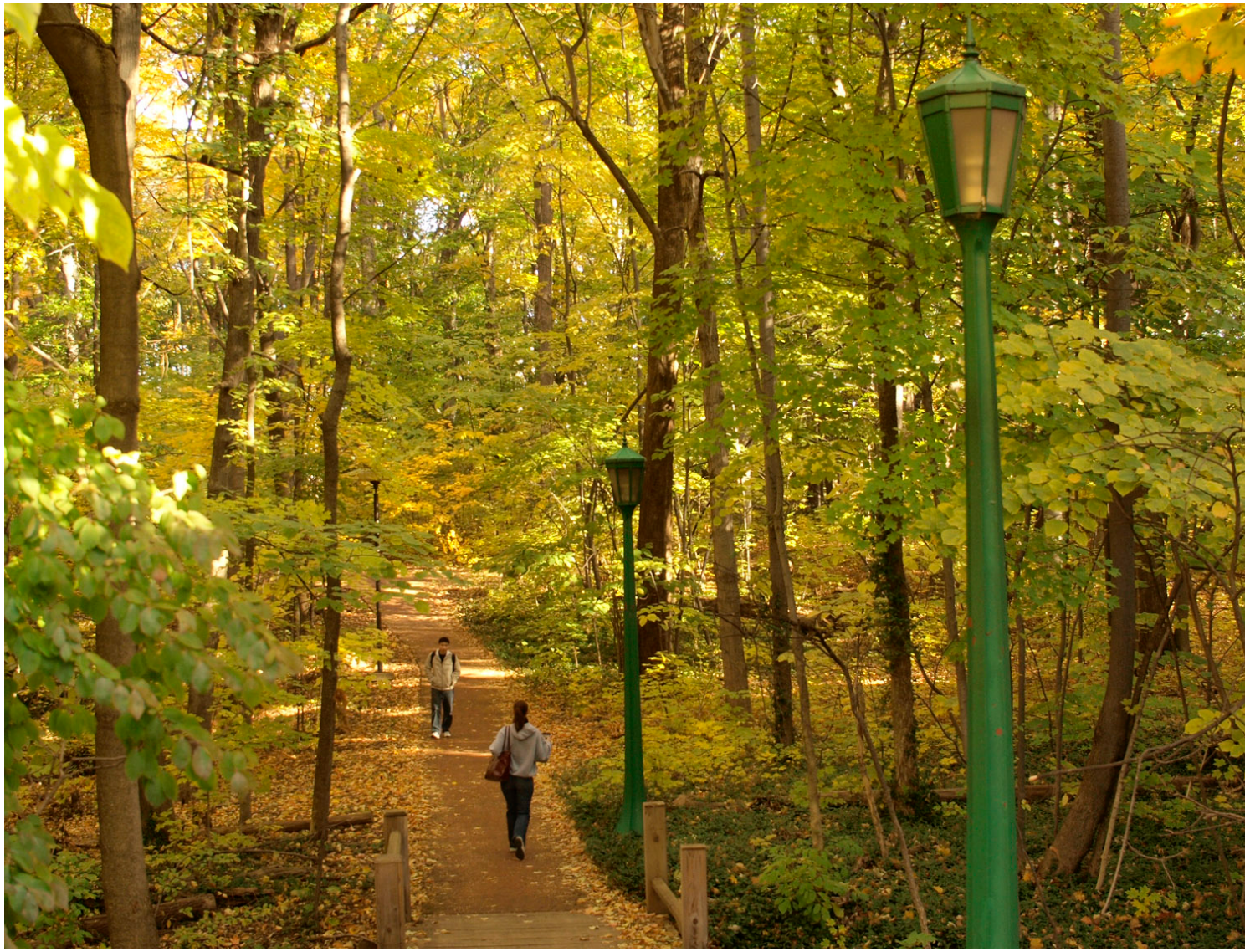
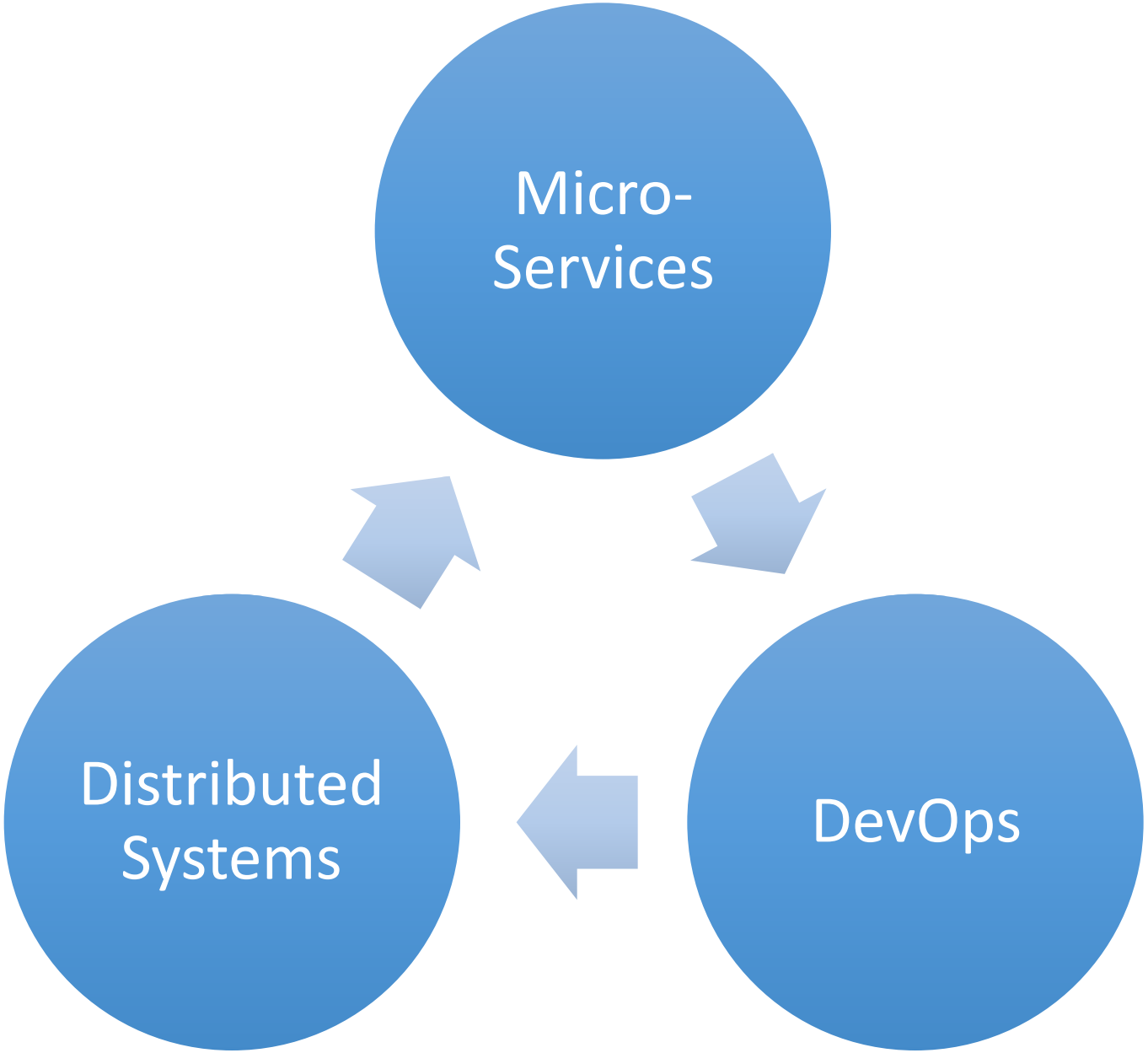
Microservices and Science Gateways

The Nature of the Class

- We'll tell you "why" and "what"
- You'll figure out "how" in the project assignments.




Micro-
Services
DevOps
Distributed
Systems

Martin Fowler's Definition of MicroServices

- Develop a single application as a suite of small services
- Each service runs in its own process
- Services communicate with lightweight mechanisms
  - "Often an HTTP resource API"
  - But that has some problems
  - Messaging and hybrid approaches
- These services are built around business capabilities
- Independently deployable by fully automated deployment machinery.
- Minimum of centralized management of these services,
  - May be written in different programming languages
  - May use different data storage technologies.

http://martinfowler.com/articles/microservices.html

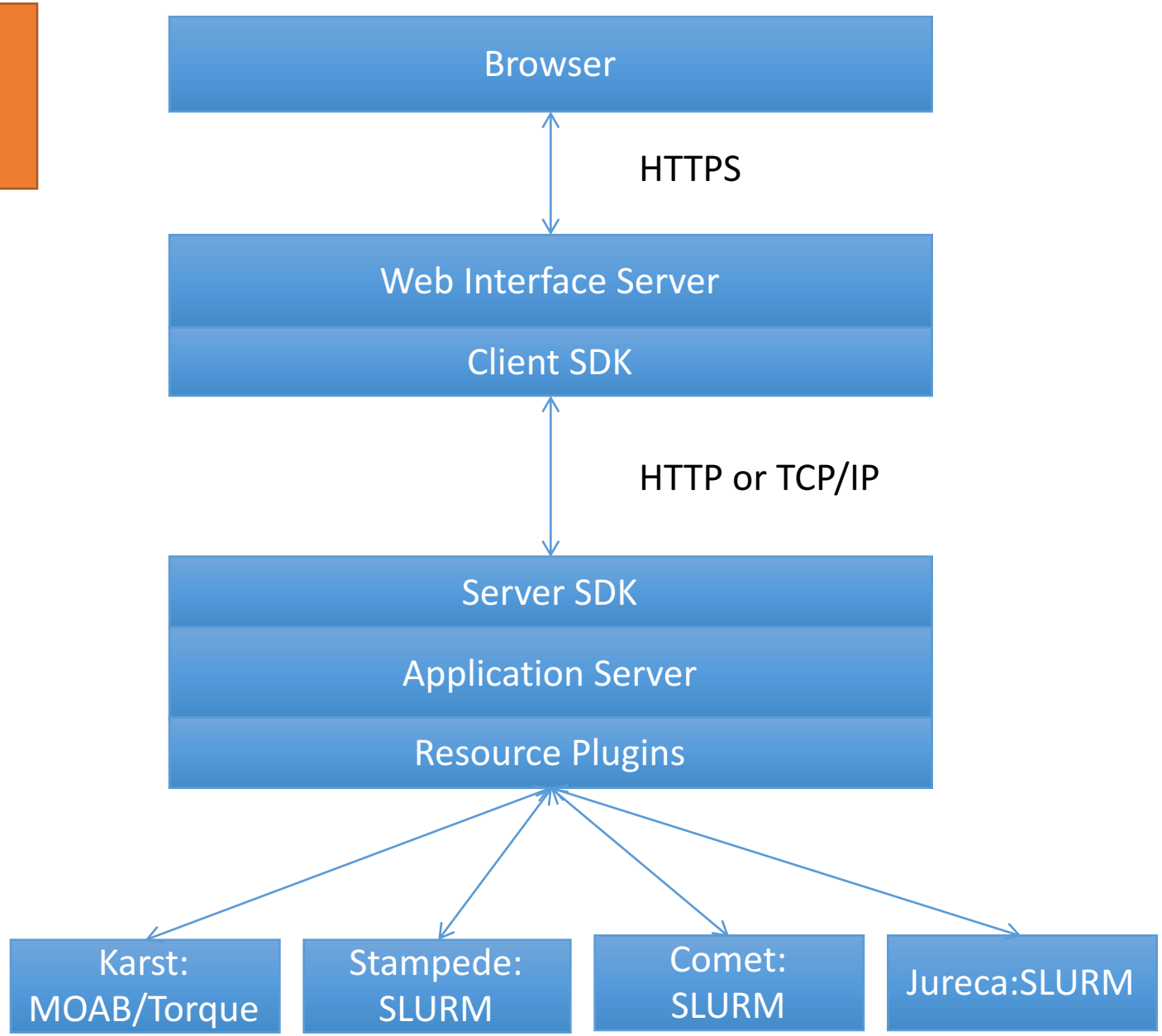
Monolithic Applications: Traditional Software Releases

- Software releases occur in discrete increments
- Software runs on clients' systems
- Releases may be frequent but they are still distinct
  - Firefox
  - OS system upgrades
- Traditional release cycles
  - Extensive testing
  - Alpha, beta, release candidates, and full releases
- Extensive recompiling and testing required after code changes
- Code changes require the entire release cycle to be repeated

MicroServices: Software as a Service

- Does your software run as a service?
- Do you run this service yourself? Or does another part of your organization run the service that your team develops?
- Traditional release cycles don't work well
  - May make releases many times per day
  - Test-release-deploy takes too long
- You can be a little more tolerant of bugs discovered after release if you can fix quickly or roll back quickly.
- Get new features and improvements into production quickly.

A Monolithic Science Gateway



Monolithic Applications

Enterprise applications three tiered architecture

- a client-side user interface
- a database
- a server-side application.

Server-side Application

- handle HTTP requests,
- execute domain logic,
- retrieve and update data from the database,
- select and populate HTML views to be sent to the browser.

This server-side application is a *monolith*

- A single logical executable.
- Changes require building and deploying a new version of the server-side application.

Other Questionable Monolithic Patterns

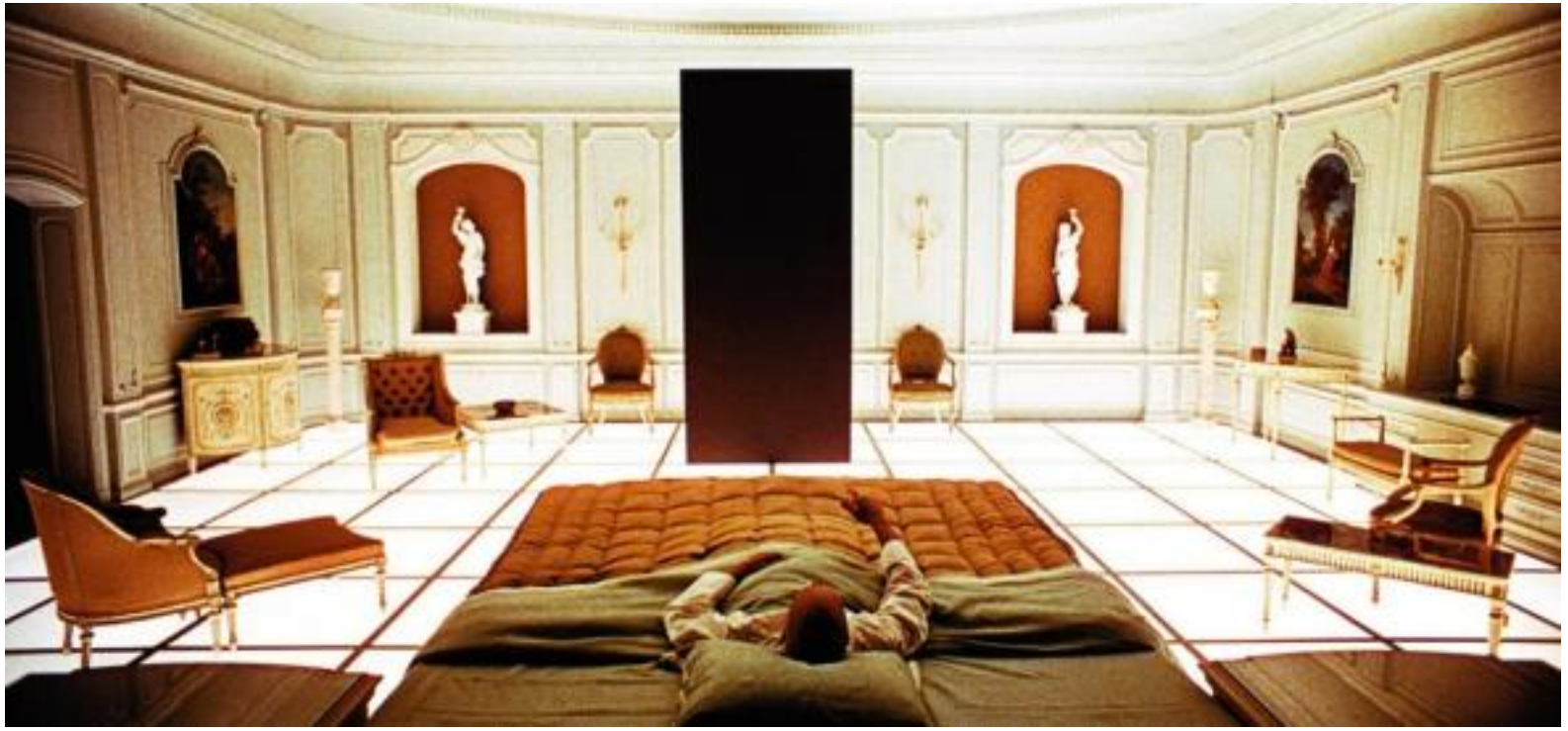
- The server-side application interacts with a single database
  - Not everything needs to fit into one DB system
  - Not just CAP choices: use the DB that fits each subsystem's problem and your development patterns
- Side effect: splitting teams into tiers
  - The DB team, the UX team, the Server App team
  - The deployment and operations team

CAP: Consistency-Availability-Partition Tolerance

Are Monolithic SaaS Applications Bad?

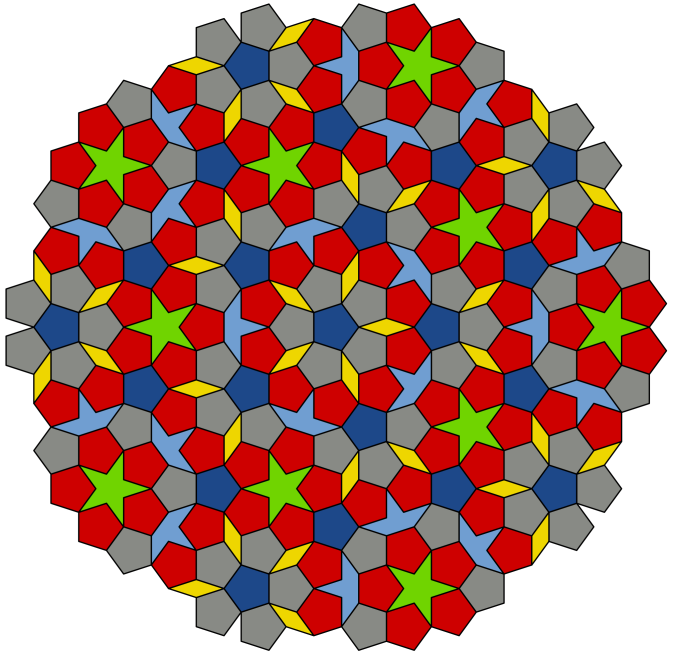
- A monolithic server is a natural way to approach building many systems.
- Monolithic applications can and should still be modular
- All your logic for handling a request runs in a single process
- Monoliths can still use DevOps
  - You can run and test the application on a developer's laptop,
  - Use a deployment pipeline to ensure that changes are properly tested and deployed into production.
- You can horizontally scale the monolith by running many instances behind a load-balancer.
- DB CAP problems don’t hit most applications

Let’s Break Up the Monolith

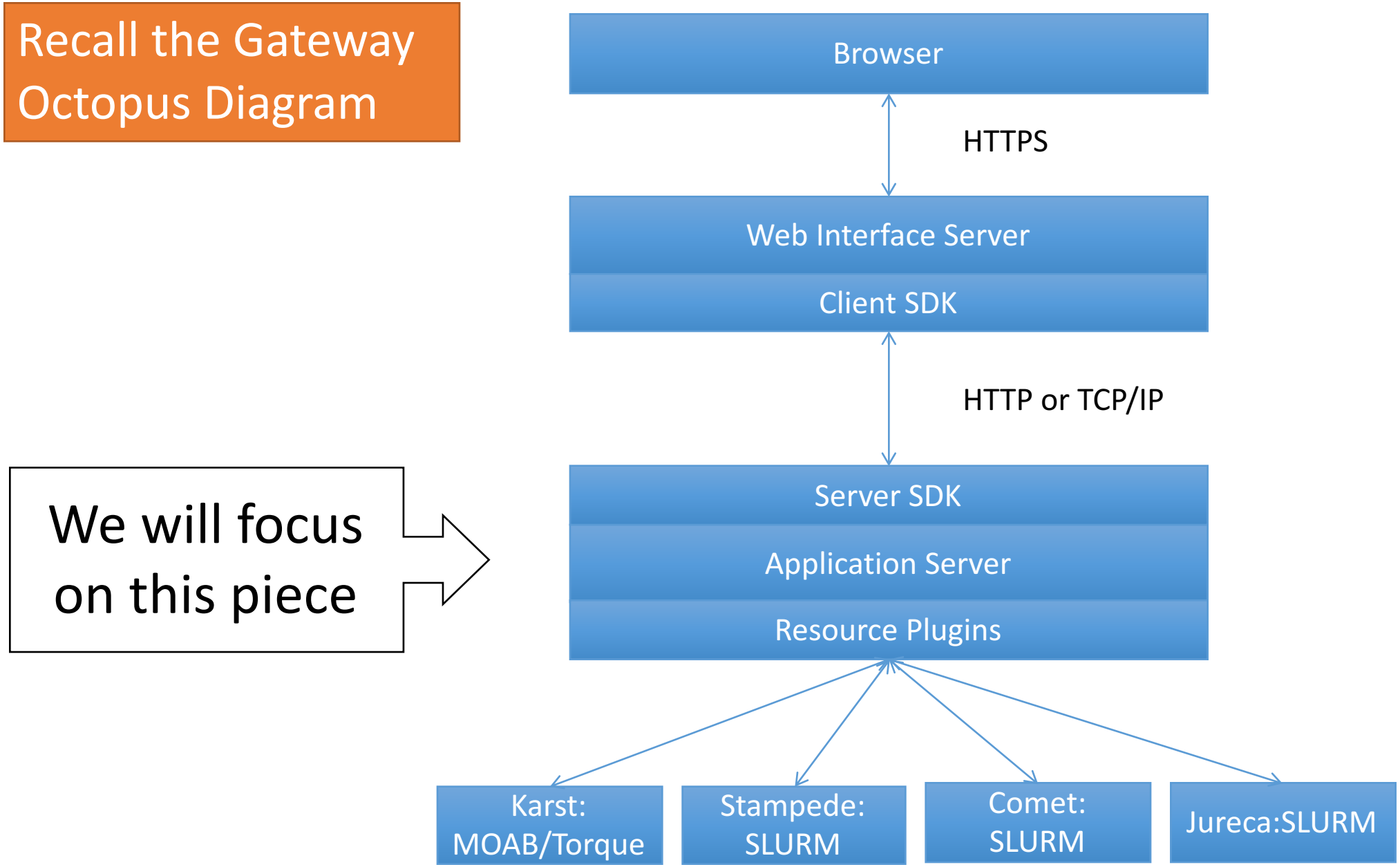


Microservices: Start with Modularity

- Think about your application.
- How can you break it into modular component parts?
- What are the interfaces? What are the messages that you need to send between the parts?
- Your code should be modular anyway.
  - OGSI and Spring do this for single JVM Java applications
  - You can maintain a module without touching other code.

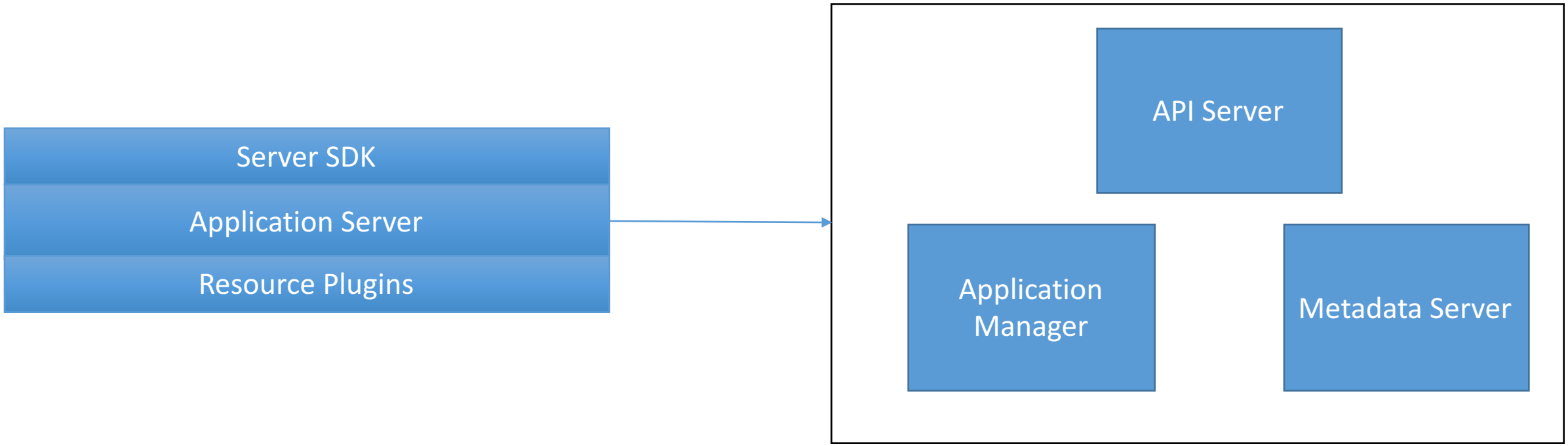


Recall the Gateway Octopus Diagram

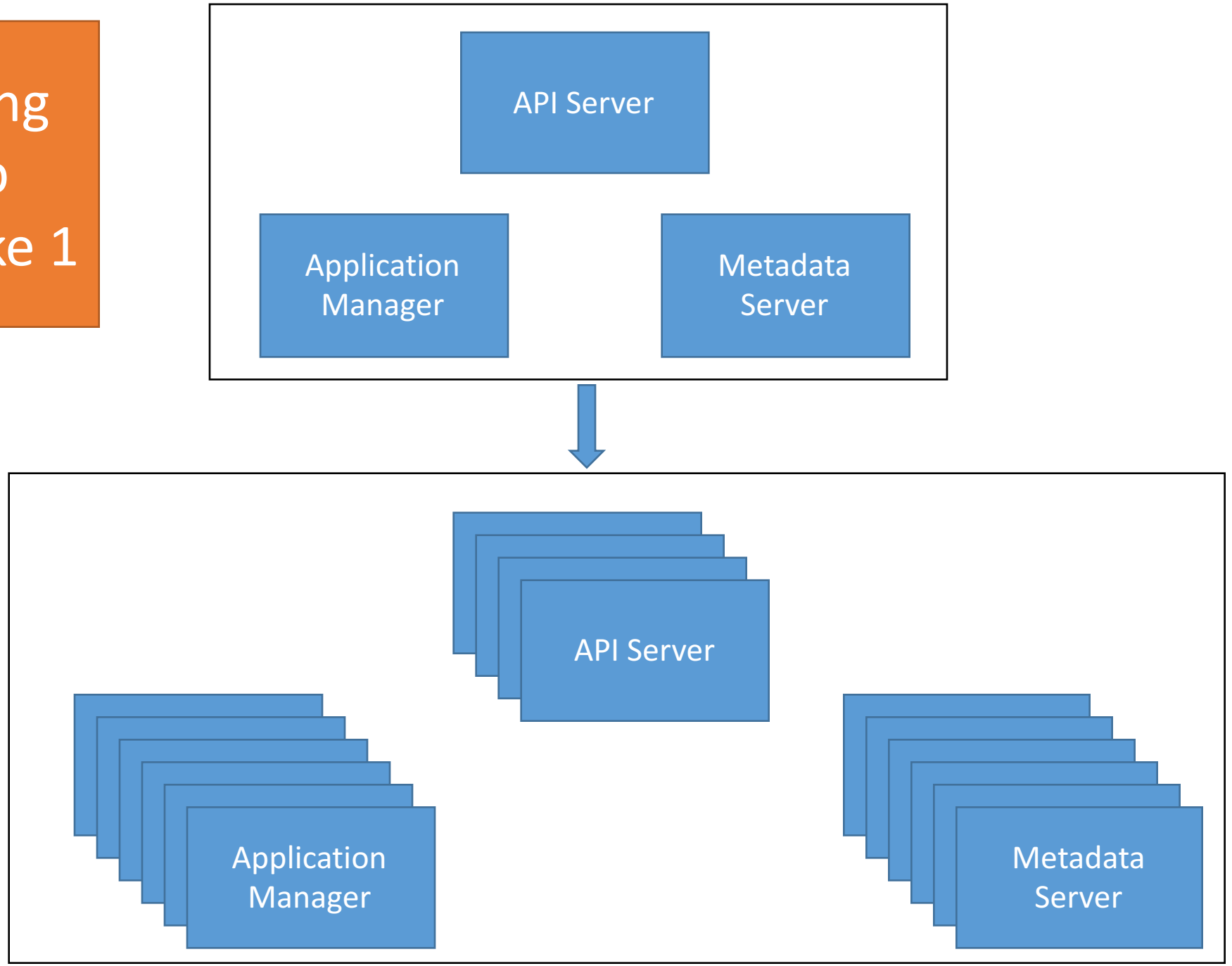


We will focus on this piece

Basic Components of the Gateway App Server



Decoupling the App Server, Take 1



What Science Gateway Microservices Can We Identify?

Thought Exercise

- Assume Amazon Cloud and a big budget
- How do these components interact?
- How can I distribute copies of components across multiple availability zones?
- What kinds of failures can occur?
- How can I handle failures of different components?

A Possible Gateway Microservice List (1/2)

Entry Point	• API Server • High availability, load-balanced
Orchestrator	• Scheduler • Highly available leader
Stateless Metadata Server	• Resources, Applications • Highly available?

A Possible Gateway Microservice List (2/2)

Stateful Metadata Server	• Experiment state
Application Manager	• Submits jobs • Fire and forget
Monitor Service	• Monitors jobs on remote resources
Logging	• Centralize your service logs

How Big Is a Microservice?

How little is micro?

Rules of Thumb You Might Encounter

- Amazon's two-pizza rule
  - Teams shouldn't be larger than what two pizzas can feed
- "A good programmer should be able to rewrite a microservice in a couple of weeks"
- Bounded Contexts from Domain Driven Design

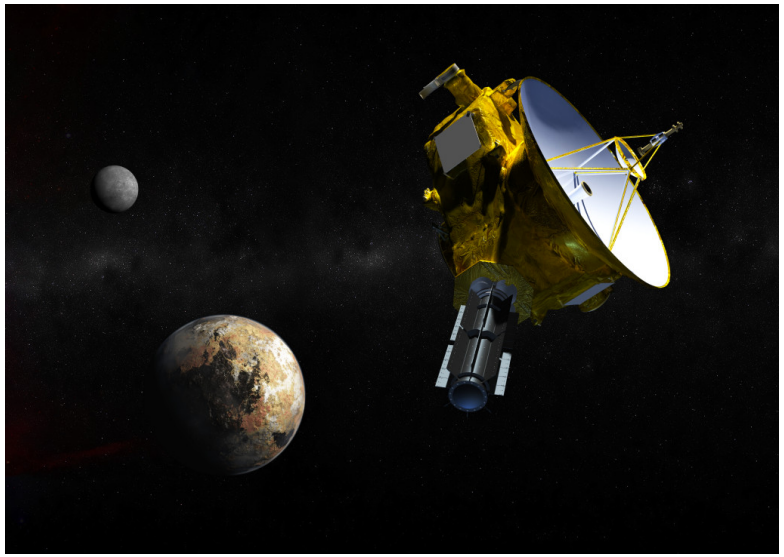
Can We Make This More Rigorous?

How Big Is a Service?

- One answer: the optimal message pattern should not change when going from single JVM to distributed services.
  - Telescope in and telescope out
- Components that run in the same process can be chatty and send lots of messages.
  - Messages can be pointers to shared data objects
  - Messages are always received
  - This is true for both request and push messaging.
  - Microsecond latencies

Pan-Galactic Components

- Components that run across the universe need different messages, either infrequent or terse.
  - No shared data objects to point to.
- Distributed components must also assume a wide variety of message failures
  - Lost connections
  - Recipients never received the message
  - Recipients received messages in the wrong order
  - Recipients crashed while processing
  - …
- Microservices are distributed systems
  - Should know and use DS ideas



How do services communicate?

- Ideally, always do the same thing.
  - Your development environment should be a miniature version of your deployment environment: containerization
  - Encapsulate and abstract
  - This touches DevOps CI/CD, so those are separate lectures
- Keep the communication patterns the same
  - Push, pull
  - 1-1, or N-N
- We'll look at messaging systems in detail in future lectures
  - Push messaging, many-to-many, reliable delivery, replay, etc

Messaging patterns between services

Request-response	• REST • Common in Internet applications but you would never do this within code in a single JVM
Push	• Common in single JVM application: Observer, listener patterns • But not so common in decoupled applications. • More generally, asynchronous messaging is a little scary.
One-to-one	• The component knows which component needs to get the new information.
Many-to-many	• Components don’t need to know which other components needs the information. They just broadcast it.

Decentralized Data Management

This is an open challenge for Apache Airavata

- The Registry component, backed by a single DB system, holds all the data

Microservice Antipattern

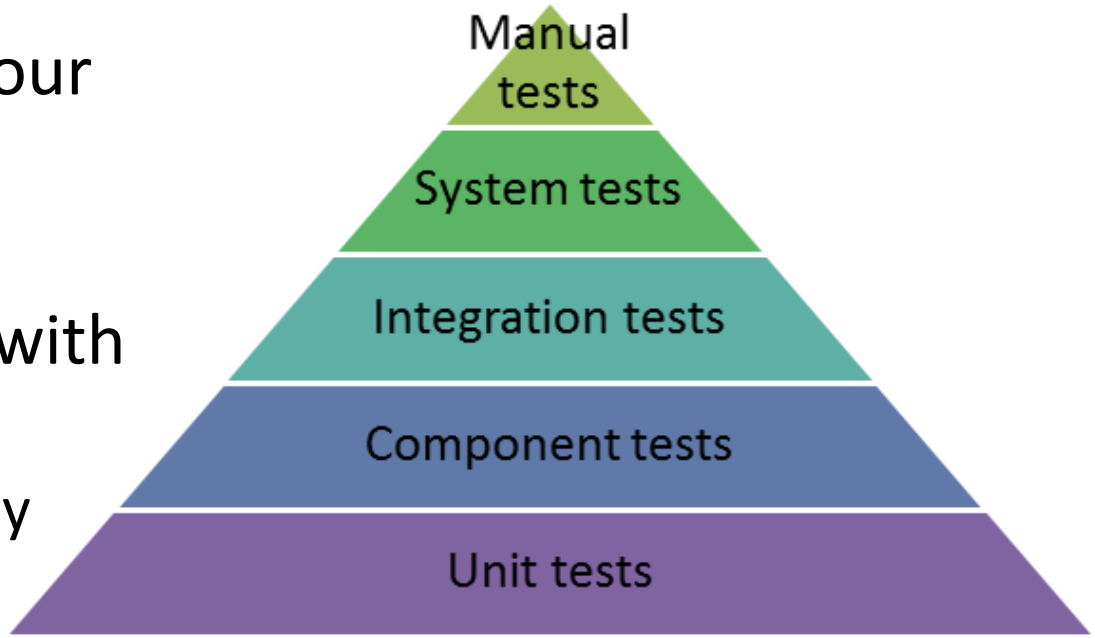
- One database has all the information for the application
- Limits your choice of data stores: you have to use the same DB technology for everything.
- Microservices constantly need to make remote DB calls
- Performance and security

Setting Data Boundaries

- This is easy when the data isn't connected or lightly connected.
- But not easy to split up a traditionally designed DB schema.
- Science gateway data boundaries
  - You may want to split stateless and stateful data
- If you split even lightly connected data across services, you will need to consider consistency
  - Strong: requires distributed transactions, which can be tricky
  - Weak: eventual consistency. Can you live with this?
- **What data boundaries do science gateways have?**

If I make everything distributed, how can I test?

- Unit tests within components are simple
- But testing the entire system becomes an operational test
- This is not a bad thing if you can script your operation tests
  - And you should
- Monolith: test your code while building with Maven
  - Bring up simple test DBs like SQLite or Derby
  - One big build does everything



Microservice Testing

- Microservice: test your code in an operation-like setting
  - Don’t build the entire code every time
  - Build only the microservice you are working on.
  - So your build should be modular
- So maybe you don’t need one big build system after all
  - And maybe you don’t need one git repo for everything...
  - And maybe you don’t need one version # that covers all your code...
- So releases become incremental
- Testing, monitoring, and logging become more continuous
  - This is a touch point between Microservices and DevOps

Designing for Failure

- Microservices need to be built to tolerate errors.
- What kinds of errors will science gateway microservices encounter?
- How do you detect these failures?
- What tools do you need?
- What strategies can you use to do system testing?

Microservices Are Not Distributed Objects

- A microservice is not an object in the OO sense.
- A microservice should be a service in the SOA sense
- Recall REST architecture: services should be as idempotent as possible.
  - If you send the same message more than once, the server state after processing the message is the same.
  - Ex: If a component receives 10 identical message saying submit a job, the component should only submit the job once.
    - How do you know the message is really identical? Timestamps, other identifiers, etc
- “Shared nothing” between services
- Message processing should have no side effects that need to be exposed to other components
  - Anti-pattern: all those output files generated by scientific applications

Some Meta-Computer Science Thoughts

Should a person or group “own” a subset of modules or microservices?

For small teams, NO in my experience.

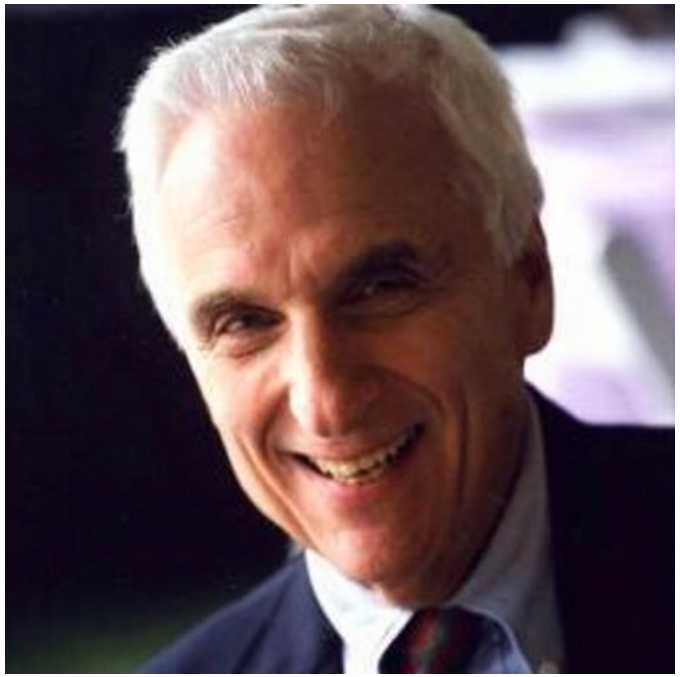
- Team members come and go
- They aren’t always available anyway
- Work-life balance will eventually become important to most
- No one should be indispensable

Module ownership leads to other problems

- “Possessive ownership” culture
- Opaque code
- “Not my problem” if it happens in another module

A DevOps Digression: Ownership and Conway's Law

- *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. –Melvin Conway*



In Other Words

- **If** you separate your organization into
  - Systems specialists,
  - DB admins,
  - Software developers,
  - User experience specialists, etc,
- And ask them to build something,
- **Then**
  - the system will look like the org chart.
- This is the opposite of the DevOps approach
- Creates the bureaucratic infighting that plague organizations.

Products, Not Projects

- The alternative: organize around products
  - A product team has all the expertise it needs
  - Jim Gray interviews Werner Vogels: https://queue.acm.org/detail.cfm?id=1142065
  - “You build it, you own it”
- Problems with this approach:
  - Someone may work on multiple products
  - Who is that person’s boss?

My Advice: Own the Entire Product

Even if you aren’t the manager. Own the product.