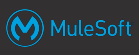
Anypoint Platform Architecture: Application Networks

Module 9
Transitioning Into Production

Objectives

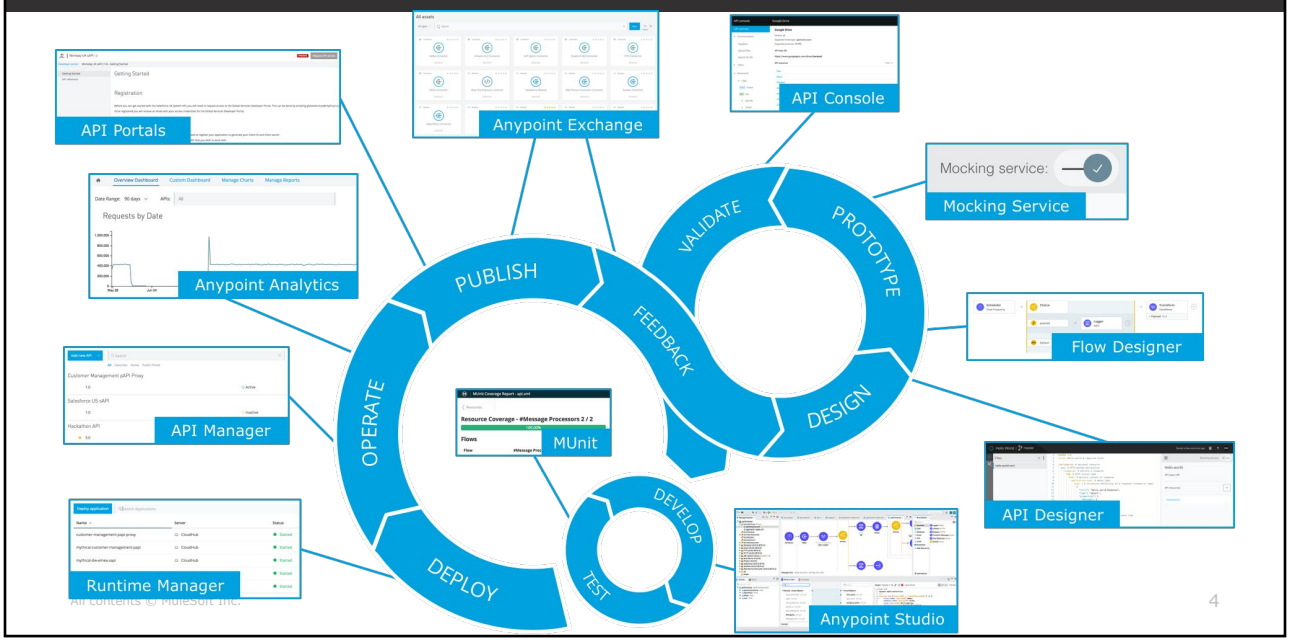


- Locate API-related activities on a **development lifecycle**
- Interpret **DevOps** using Anypoint Platform tools and features
- Design **automated tests** from viewpoint of API-led connectivity
- Identify the factors involved in **scaling** API performance
- Use **deprecation and deletion of API** versions
- Identify single **points of failure**

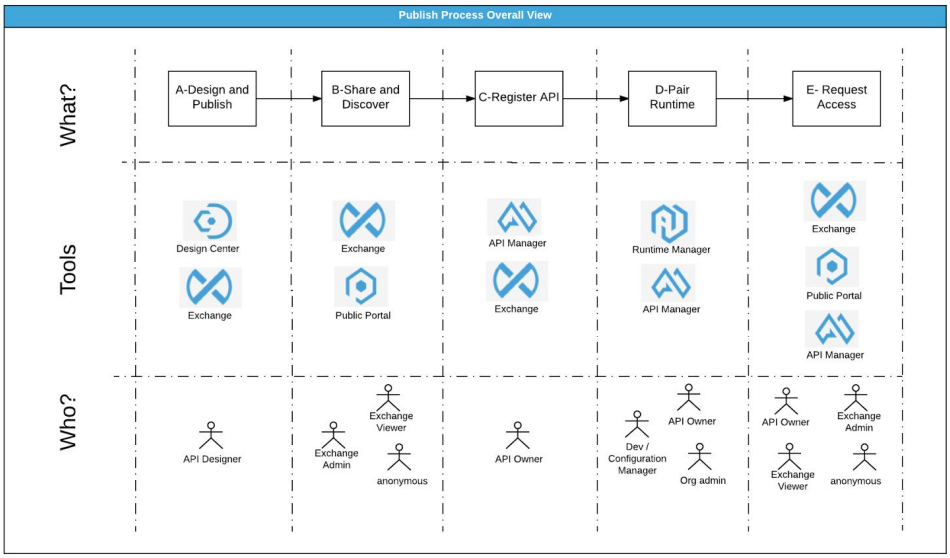
Understanding the development lifecycle and DevOps

Keeping the development lifecycle in perspective

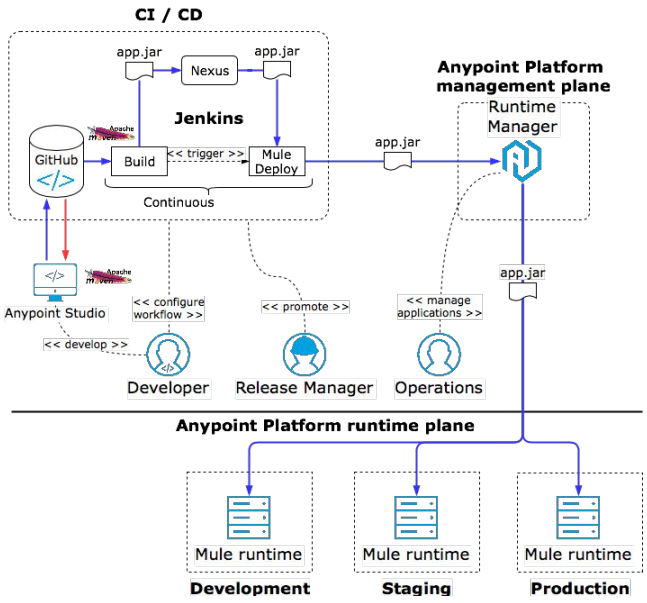
MuleSoft



Keeping the development lifecycle in perspective



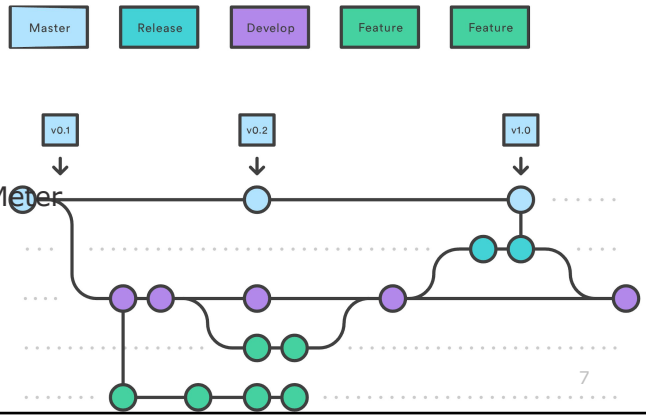
Building on a strong DevOps foundation



Building on a strong DevOps foundation

MuleSoft

- **RAML** definitions and fragments in artifact repository: **Exchange**
- **Source code**: one **GitHub** repo per API implementation
- Develop on **feature branches** off the develop branch (GitFlow)
- **Developers** implement Mule apps and all automated tests
  - Unit, integration, performance
  - **Studio**, JUnit, **MUnit**, SOAPUI, JMeter
  - Maven, **Mule Maven plugin, MUnit Maven plugins**, ...



Building on a strong DevOps foundation

MuleSoft

- Developers submit GitHub **pull requests**
- **Code review** of the pull request
  - If OK and all tests pass: **merge** the pull request into develop branch
- Triggers **CI** pipeline:
  - **Jenkins** delegating to Maven
  - Compiled, packaged, **unit-tested** (embedded Mule runtime)
  - Deployed to an **artifact repository**
    - Private **Nexus**
- After sufficient features cut a **release**:
  - Tag, create release branch and ultimately merge into master branch

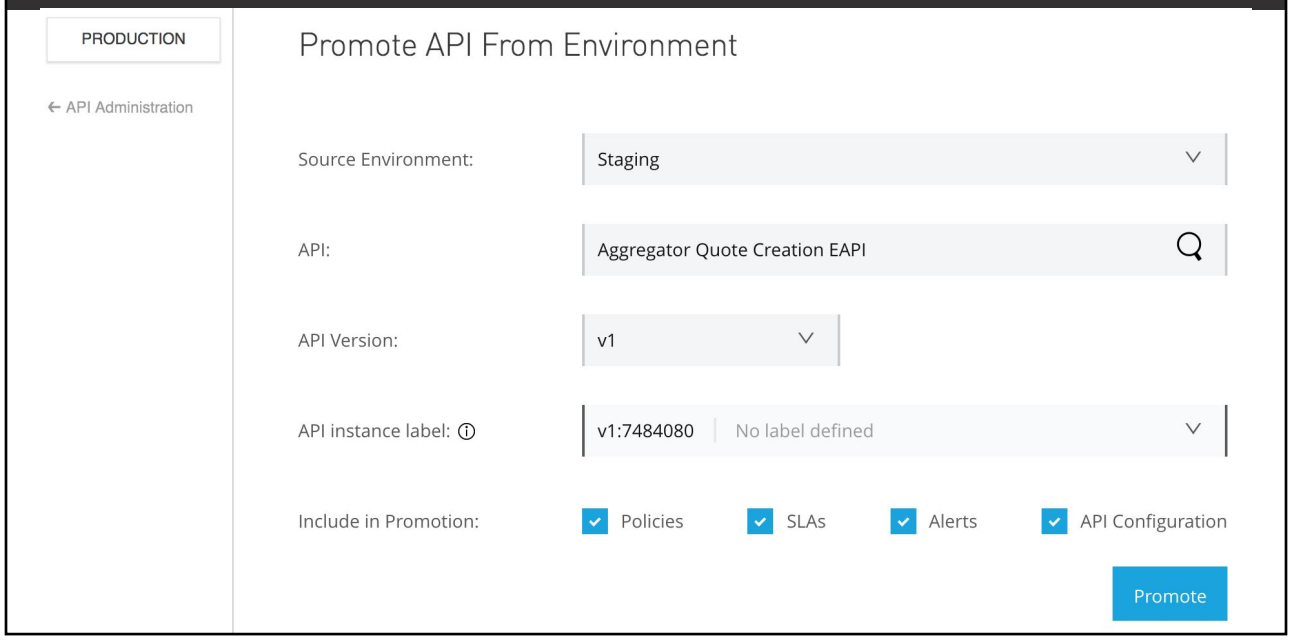
Building on a strong DevOps foundation

- Triggers **CI/CD** pipeline:
  - **CI** pipeline is executed as before:
    - Compiled, packaged, **unit-tested**
    - Deployed to **artifact repository**
  - Automatically and/or through manual trigger
    - Well-defined version of API implementation **retrieved** from artifact repo
    - Deployed into **staging** environment
    - **Integration** and performance tests run over HTTP/S
    - Deployed into **production** environment
    - "**Deployment verification** sub-set" of the functional end-to-end tests is run
      - On failure **rollback**: immediate execution of the CD pipeline with the last good version

Promoting APIs and API implementations to higher environments

- **API Manager** supports promoting parts of an API instance
  - Does not copy **API clients**
    - Promoted APIs start off without any registered API clients
  - Share **"Implementation URL" and "Consumer endpoint"**
    - Change after promotion
- Similarly, **Runtime Manager** can "deploy from Sandbox"
- **Automate** via Platform APIs and/or CLI
  - May integrate into **CI/CD**

Promoting an API instance to a higher environment



Designing automated tests for APIs, API implementations and the application network

Understanding API-centric automated testing

- Testing **does not change** fundamentally
  - Same types of tests and activities of preparing for, executing and reporting on tests are still **applicable**
  - Here: only addresses a few **selected topics**
- **APIs and API specifications** take center stage when testing application networks
- Distinction between **unit tests and integration tests**
- **Resilience tests** are important
  - Establish confidence in the **fail-safety** of the application network

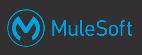
Unit versus integration tests

- **Unit** tests
  - From **embedded Mule** runtime
  - Implemented using **MUnit**
  - Read-only invocations of APIs/systems: invoke **production endpoints**
  - Write interactions: implement **mocks** using MUnit
  - White-box
- **Integration** tests
  - Invoke API just **like in production**: over HTTP/S with API policies applied
  - Deployment into **staging environment**, with all **dependencies** available
  - **No mocking**
  - Implemented using **SOAPUI** and its **Maven plugin**
    - Trigger API invocations and assert responses
  - Black-box

Designing automated API-centric integration tests

- Test scenarios for each **API**
- **Functional and non-functional** tests, incl. performance tests
- Test scenarios driven by the **API specification**
- Just use information from **Exchange**
  - Ignore actual **API implementation**
  - **Highlights deficiencies** of the API's discoverable **documentation**
  - All interactions in **API Notebook** covered by a test scenario
- Automatically execute tests: invoking API **endpoint** over HTTP/S
- Test assertions must go beyond (but include) adherence to the **RAML definition** in terms of data types, media types, …
- Must execute in special production-like **staging environment**
  - **Production-safe** sub-set as "deployment verification test suite"

Unit testing API implementations



- API implementations have many and **complex interactions** with other systems
  - Makes unit testing **difficult**
- **MUnit**
  - Specifically intended for unit testing **Mule applications**
  - Can **stub**-out external dependencies of a Mule application
  - Dedicated IDE-support in **Studio**
  - Can be invoked from **Maven builds**

Testing the resilience of application networks

- Web of highly **interconnected APIs**
- Resilience testing **disrupts** that web
  - Asserts that the resulting **degradation** is within **acceptable** limits
- **Important practice** in the move to application networks
- Possible **automated** approach:
  - Software tool similar in spirit to **Chaos Monkey**
  - Acts as API client to the **API Platform API**
  - Automatically adds, configures and removes **custom API policies** on APIs
    - Erratically throttle or **interrupt invocations** of APIs to which they are applied
  - Normal automated **integration tests** executed alongside

Exercise: Reflect on resilience testing

In your experience with testing complex distributed systems:

1. Is resilience testing an established part of organizations' testing strategy?
2. Should resilience or performance tests be run against the production environment?
3. Does focusing resilience testing on API invocations make this a more approachable practice?
   a. Does it reduce the effectiveness of resilience testing compared to more general resilience testing approaches?

End-to-end test scenarios for Acme Insurance

MuleSoft

- Test cases should be **executed**
  - With the application network in **healthy** state
  - While **resilience tests** are disrupting application network
- Small **selection** for "Aggregator Quote Creation EAPI":
  - Invoke with valid, invalid and missing policy description from Aggregator
  - Invoke for existing and new policy holder
  - Invoke for policy holder with a perfectly matching in-force policy
  - Invoke at 500, 1000 and 1500 requs/s
  - Invoke over HTTP and HTTPS
  - Invoke from API client with valid, expired and invalid client-side certificate

End-to-end test scenarios for Acme Insurance

MuleSoft

- Small **selection** for "Motor Policy Holder Search SAPI":
  - Invoke with valid, invalid and missing search criteria
  - Invoke for search criteria matching 0, 1, 2 and almost-all policy holders
  - Invoke for policy holder with only home and no motor policies
  - Invoke at 500, 1000 and 1500 requs/s
  - Invoke with valid and invalid client token and without client taken
- For "Motor Claims Submission PAPI":
  - Invoke polling endpoint once per original request, 1000 times per second, not at all

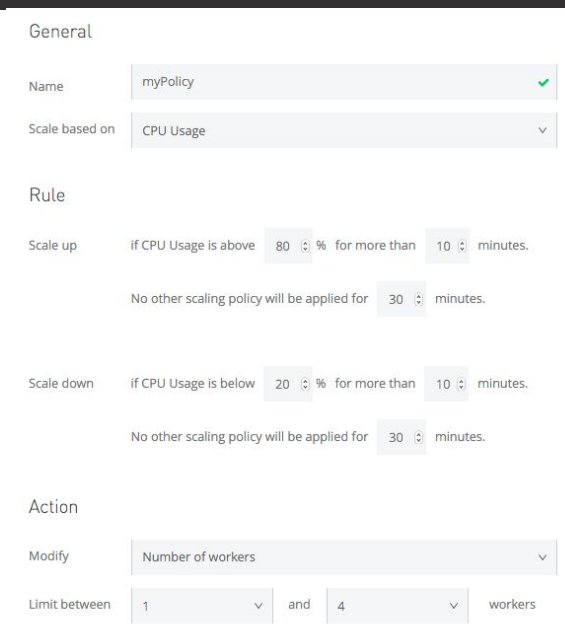
Scaling the application network

Ways to scale the performance of an API

MuleSoft

- Two main ways of scaling the performance of an existing API and implementation:
  - **Vertical** scaling
    - Scaling **performance of each node** of API implementation / proxy
    - CloudHub: **worker sizes**
  - **Horizontal** scaling
    - Scaling the **number of nodes**
    - CloudHub et al.: **scaleout and load balancing**, up to 8 workers
- API proxies **scaled independently** of API implementations
  - **Realistic**: more/larger instances of API implementation than API proxy

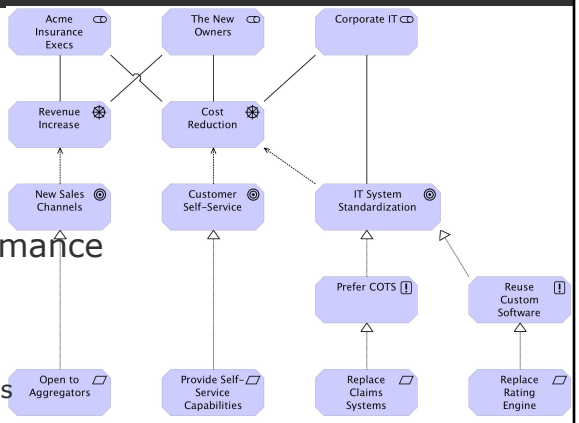
Autoscaling CloudHub workers



An API scales for its API clients

- An API's reason for existence is to be consumed by **API clients**
  - **QoS and performance** must suit these API clients
  - Also **change** in performance over time
- API implementation team must understand **projected performance needs** of all its API's clients
  - Must be prepared to scale their API's performance to **meet those needs**
  - May require **re-evaluating QoS guarantees and SLAs** with all **API dependencies**

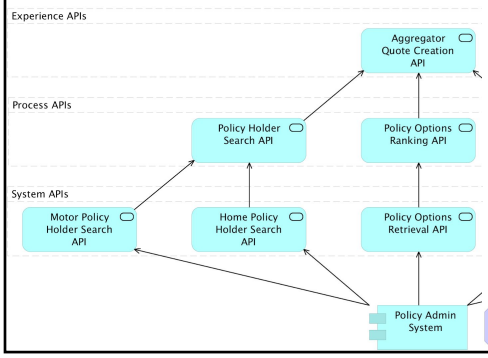
The C4E owns systemic performance considerations



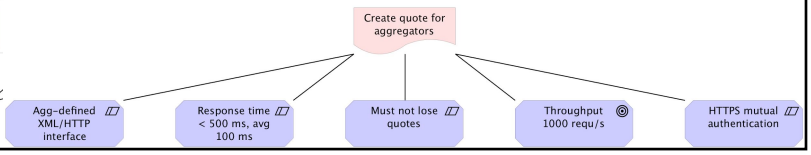
- APIs exist for **business goals**
- API performance **supports** these
  - Must be **balanced**
- **Systemic** perspective on API performance
  - Application network-wide analysis
    - **Dependencies** between APIs
    - **Business goals** to which API contributes
    - Relative **importance** of goal
- Responsibility best assigned to **C4E**
- **Data** available through Anypoint Platform APIs

Exercise: Scaling "Policy Options Retrieval SAPI"

"Policy Options Retrieval SAPI" has been deployed to one 1 vCore CloudHub worker with API policies, but does not deliver the required throughput. Suggest an ordered sequence of steps that could be taken to improve throughput. Use your experience and intution to guide your assumptions!

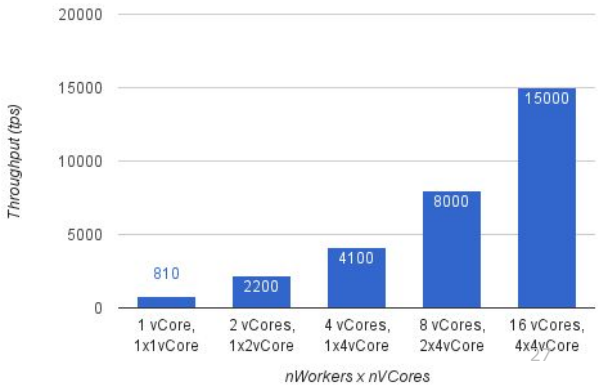


>	IP whitelist	Security	IP filtered
>	Rate limiting - SLA based	Quality of service	SLA Rate Limiting, Client ID required
>	Spike Control	Quality of service	Baseline Rate Limiting



Solution: Scaling "Policy Options Retrieval SAPI"

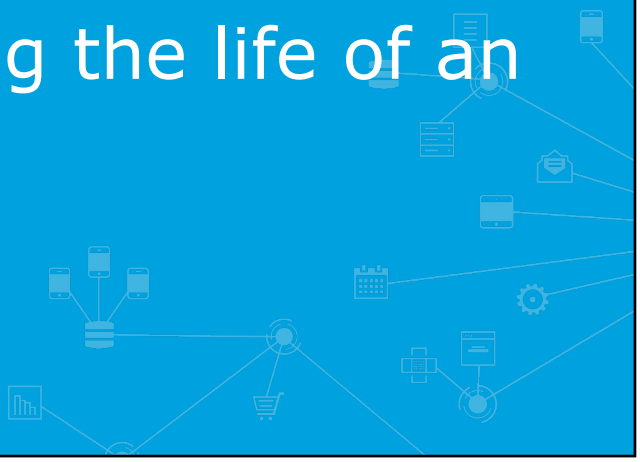
- **5000 requs/s**
- **More than 4 vCores** needed
- **Policy Admin System** (PAS) incapable
  - **Caching in memory**
  - As **few CloudHub workers** as possible
  - **Recurring input to SAPI** should result in high cache hit rate



Solution: Scaling "Policy Options Retrieval SAPI"

1. **Increase num workers** for SAPI >= 2 until **PAS is bottleneck**
   a. Adjust **Spike Control** API policy to **protect PAS**
   b. Assumption: **two 1 vCore** workers sufficient
2. **API proxy** in front of SAPI with **caching API policy** and SLA-based Rate Limting
   a. Deploy to **two 4 vCore** workers
3. Scale to **two 8 vCore** workers: improves cache hit rate
4. Try **one 16 vCore** worker (with auto-restart!): maximizes cache hit rate at expense of HA
   a. "Policy Options Ranking PAPI": cient-side caching, static fallback results

Gracefully ending the life of an API



End-of-live management on the level of API version instead of API implementation

MuleSoft

- **API client**
  - **Sends** API invocations to endpoint of API
  - **Receives** responses in accordance with contract and expected QoS
- Requires an **API implementation**
  - But API client is **unaware** of API implementation itself
- API implementation can be **changed** without alerting API clients
- All **incompatible changes to the API**, its contract or promised QoS, must be communicated to all API clients
  - Introduce of a **new version** of the API
  - Subsequent phased **ending of live of the previous version**

Deprecating and deleting an API version on Anypoint Platform

MuleSoft

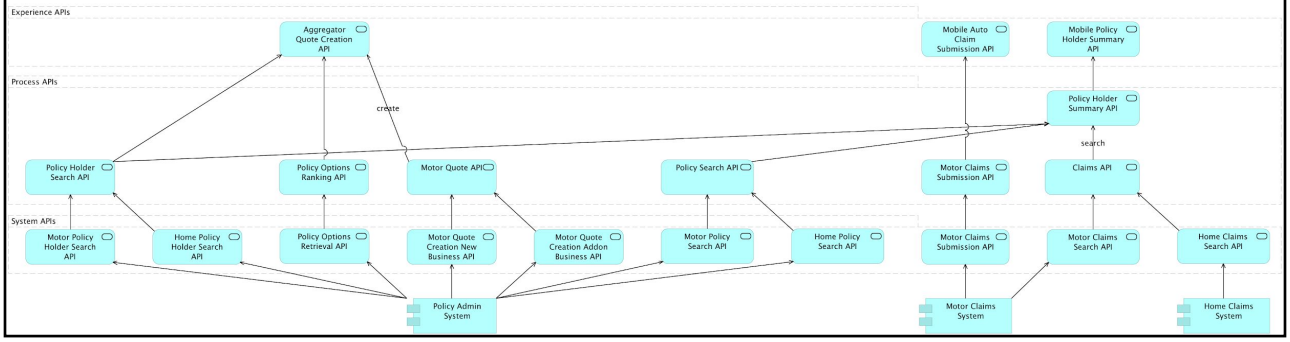
- **API Manager**
  - **Deprecate** API **instance** in environment
  - **Prevents** API consumers from requesting **access** - then delete
- **Exchange**
  - **Deprecate** indivual **asset version**
  - **Informs** API consumers, does not prevent requesting access
  - Entire Exchange entry deprecated if all asset versions deprecated

Identifying points of failure

Exercise: Identify points of failure in Acme Insurance's application network

Assume a deployment of all APIs in Acme Insurance's application network to a MuleSoft-hosted Anypoint Platform using CloudHub:

1. Identify **points of failure** in this architecture
2. Are there any components that are not redundant, i.e., that constitute **single points of failure**?



Exercise: Identify points of failure in Acme Insurance's application network

- **Every node and system** is potential point of failure
  - Failure of API Manager:
    - Already-applied **API policies** continue being in force
    - New Mule runtimes not functional until they can **download API policies**
    - Eventual **overflow of buffers** that hold undelivered API analytics events
- **True** single points of failure
  - API implementations deployed to **1 CloudHub worker w/o auto-restart**
  - **AWS region** for control plane and runtime plane (CloudHub workers)
  - **Home Claims System** ?
  - Every **new deployment** of an API implementation constitutes a single point of failure **for all its API clients**

Summary

Summary

MuleSoft

- API definition, implementation and management can be organized along an **API development lifecycle**
- **DevOps** on Anypoint Platform builds on and supports well-known tools like Jenkins and Maven
- API-centric **automated testing** follows standard testing approaches with emphasis on **integration and resilience tests**
- **Scaling API performance** must match the API clients' needs
  - Requires the **C4E**'s application network-wide perspective
- Gracefully **decommission API versions** using deprecation
- Anypoint Platform has no inherent **single points of failure** but every deployment of an API implementation can become one