By standardizing the format of change information using JSON and automating the process of gathering this information from cloud service providers, you can improve the visibility and management of changes in your cloud environment.

However, the JSON format and script should be refined and tailored to fit your specific requirements and the structure of the release notes provided by the cloud service providers. Furthermore, you'll need to develop a more comprehensive system that can:

1. Automatically fetch and process change information from multiple cloud service providers.
2. Incorporate the extracted data into your organization's change management processes, enabling you to assess the impact of changes on your security controls, compliance, and overall risk posture.
3. Notify relevant stakeholders of upcoming changes, providing them with adequate time to review and prepare for these changes.
4. Continuously monitor your cloud environment and configuration settings to ensure that your preventive controls remain effective and are not accidentally overridden by changes made by the cloud service providers.

By developing such a system, you can bridge the change management gap and help your organization proactively address potential risks and security concerns in your cloud environment. It's essential to engage in a dialogue with the cloud service providers, encouraging them to adopt standardized reporting formats and improve their change in communication practices, which will benefit all customers, regardless of their size.

Building a comprehensive system for automatically fetching and processing change information involves a combination of web scraping, API integration, data transformation, and collaboration with cloud service providers for standardized reporting formats. Here's a high-level outline of the steps to build such a system:

1. Identify the sources of change information for each cloud service provider, such as release notes, API endpoints, and newsletters.
2. For each provider, develop a method to fetch the information automatically. This can involve web scraping for release notes, API calls for structured data, or parsing newsletters using email APIs and natural language processing techniques.
3. Transform the fetched information into a standardized JSON format that includes relevant fields such as the provider, service impacted, class of impact, narrative, URL, setting changes, NDA, date of release, and impact severity.
4. Store the transformed data in a database or data lake, which allows for efficient querying and analysis of the change information.
5. Create a notification system to alert relevant stakeholders about upcoming changes, giving them enough time to review and prepare for potential impacts.

6. Develop an interface or dashboard to visualize and manage the change information, allowing your organization to assess the impact of changes on your security controls, compliance, and overall risk posture.
7. Continuously monitor your cloud environment and configuration settings to ensure that your preventive controls remain effective and are not accidentally overridden by changes made by the cloud service providers.
8. Engage in a dialogue with the cloud service providers to encourage them to adopt standardized reporting formats and improve their change communication practices.

The implementation details, programming languages, frameworks, and tools used to build such a system will depend on your organization's specific requirements, available resources, and existing infrastructure. The Python programming language, along with libraries like Requests, BeautifulSoup, and various API libraries, can be a good starting point for fetching and processing change information. For storing and analyzing the data, you can use databases like PostgreSQL or data lakes like AWS S3 or Google Cloud Storage, and data processing frameworks like Apache Spark or Dask.

For the frontend, you can use web development frameworks like React or Angular to build a dashboard that visualizes the change information and allows for efficient change management.

*Note: There would be a requirement to build a front end GUI for end users to visualize this data for ease of use.

#NOTE: These items can all be tailored to fit the type of data we want to receive via API or adjust for web scraping tools.

```
{
  "Service Impacted": ["S3"],
  "Class of Impact": [
        "Performance",
        "Availability",
        "Scalability",
        "Compliance",
        "Compatibility",
        "Data Management",
        "Security",
        "Cost",
```

```
            "Functionality",
            "Integration",
            "User Experience",
            "Documentation"
  ],
  "Narrative": "Oops, I accidentally opened the whole thing",
  "URL": "https://blogpost.htm",
  "Setting": "GetObject",
  "Setting from": "None",
  "Setting to": "ANY",
  "NDA": ["Yes", "No"],
  "Date of release": "2023-04-04T00:00:00",
  "Impact Severity": "High"
}
```


To provide an example of data extraction and transformation, let's assume the cloud provider API returns release notes data in the following format:


```
{
  "results": [
            {
            "service": "S3",
            "impact": ["Performance", "Cost"],
            "description": "Oops, I accidentally opened the whole thing",
            "link": "https://blogpost.htm",
            "configuration": {
            "setting": "GetObject",
            "from": "None",
```

```
            "to": "ANY"
            },
            "nda_required": false,
            "published_at": "2023-04-04T00:00:00"
            },
            {
            "service": "EC2",
            "impact": ["Security", "Compliance"],
            "description": "Updated encryption methods",
            "link": "https://anotherblogpost.htm",
            "configuration": {
            "setting": "Encryption",
            "from": "AES-128",
            "to": "AES-256"
            },
            "nda_required": true,
            "published_at": "2023-04-03T00:00:00"
            }
 ]
}
```


We can then modify the Python script to process and transform the data into the desired JSON format:

```
import requests
import json

def calculate_impact_severity(change):
```

```
        # Add your logic to calculate impact severity here
        pass

api_key = "your_api_key_here"
api_endpoint = "https://api.example.com/v1/release-notes"

headers = {
        "Authorization": f"Bearer {api_key}"
}

response = requests.get(api_endpoint, headers=headers)

if response.status_code == 200:
        api_data = response.json()

        transformed_data = []

        for release_note in api_data["results"]:
        change_data = {
        "Service Impacted": release_note["service"],
        "Class of Impact": release_note["impact"],
        "Narrative": release_note["description"],
        "URL": release_note["link"],
        "Setting": release_note["configuration"]["setting"],
        "Setting from": release_note["configuration"]["from"],
        "Setting to": release_note["configuration"]["to"],
        "NDA": ["Yes", "No"][int(release_note["nda_required"])],
```

```
        "Date of release": release_note["published_at"],
        "Impact Severity": calculate_impact_severity(release_note)
        }

        transformed_data.append(change_data)

        print(json.dumps(transformed_data, indent=2))
else:
        print(f"Error fetching data: {response.status_code}")
```

In this script, the **for** loop iterates through each release note returned by the API, extracts relevant information, and transforms it into the desired JSON format. The transformed data is then appended to a list called **transformed_data**, which holds all the changes in the required format.

Please note that this example assumes a specific structure for the API response. You'll need to adjust the extraction and transformation logic to match the actual structure of the data provided by your cloud provider API.

While the JSON format itself cannot directly connect to cloud services or track release notes, you can create a script or application that utilizes the JSON format and interacts with the cloud services and relevant data sources. Here's a high-level approach to achieve this:

1. **Identify data sources**: Collect URLs or API endpoints for release notes, announcements, or updates from each cloud service provider you use. Keep a list or a configuration file with these data sources for easy access.
2. **Fetch updates**: Write a script or application to periodically fetch new release notes or announcements from the identified data sources. This can be done using web scraping libraries like Beautiful Soup (Python) or Cheerio (JavaScript), or by utilizing cloud provider APIs if available.
3. **Extract relevant information**: Process the fetched updates to extract relevant information, such as impacted services, changes, and release dates. Convert this information into the JSON format you defined earlier.
4. **Determine impact on your deployments**: For each change represented in the JSON format, compare it with your current deployments to determine the potential impact. This

can involve rule-based analysis, machine learning techniques, or a combination of both, as previously discussed.

    1. **Create a connection into your current cloud environments to compare the impact of what the JSON file is identifying (future goal?)**

5. **Store and manage change data**: Save the JSON objects, along with their calculated impact scores or severity, in a centralized database or file storage system, making it easier to manage, query, and analyze the data.
6. **Notify relevant teams**: Set up a notification system to alert relevant teams when a change with a significant impact is detected, allowing them to take appropriate action.
7. **Continuously update**: Periodically fetch and process new updates, refining your impact analysis rules or machine learning models as needed to ensure accuracy and relevance over time.

By following this approach, you can create a system that automatically tracks release notes or announcements from your cloud service providers, converts them into the defined JSON format, and compares the changes with your current deployments to assess the potential impact. This will enable your organization to proactively respond to service or feature changes and make informed decisions about your cloud environment.

A more generalized JSON format that could be potentially achievable from multiple cloud service providers would include the most common and essential fields related to change management. Here's an example of a simplified JSON format:

```
{
  "Provider": "CloudProviderName",
  "Service": "ServiceName",
  "ChangeType": "Update|Patch|NewFeature|Deprecation",
  "Impact": {
        "Performance": "Low|Medium|High|N/A",
        "Security": "Low|Medium|High|N/A",
        "Availability": "Low|Medium|High|N/A",
```

```
        "Compliance": "Low|Medium|High|N/A",
        "Cost": "Low|Medium|High|N/A",
        "UserExperience": "Low|Medium|High|N/A"
    },
    "Description": "Description of the change and its implications",
    "URL": "https://release-notes-url",
    "EffectiveDate": "2023-04-04T00:00:00",
    "AffectedSettings": [
        {
        "SettingName": "Setting1",
        "PreviousValue": "OldValue",
        "NewValue": "NewValue"
        },
        {
        "SettingName": "Setting2",
        "PreviousValue": "OldValue",
        "NewValue": "NewValue"
        }
    ],
    "RequiresUserAction": true|false
}
```

This JSON format includes essential information about the change, such as the provider, affected service, change type, and impact across different dimensions like performance, security, availability, compliance, cost, and user experience. It also includes a description of the change, a link to the release notes, the effective date, and information about affected settings.

Additionally, a field called **RequiresUserAction** is included to indicate if the change requires any action from the customer to take effect or maintain the current system state.

This format is more generalized, and cloud providers should be able to adopt it more easily, as it focuses on the most common aspects of changes that might affect customers. Keep in mind that some providers may have more specific fields or details about their changes, so the format might need to be adjusted or extended on a case-by-case basis.

**Optimized Version to Compare: **

```
{
 "provider": "CloudProviderName",
 "service": "ServiceName",
 "changeType": "Update|Patch|NewFeature|Deprecation",
 "impact": [
        {
        "type": "Performance|Security|Availability|Compliance|Cost|UserExperience",
        "severity": "Low|Medium|High|N/A"
        }
 ],
 "description": "Description of the change and its implications",
 "url": "https://release-notes-url",
 "effectiveDate": "2023-04-04T00:00:00",
 "affectedSettings": [
        {
        "name": "Setting1",
        "previousValue": "OldValue",
        "newValue": "NewValue"
        }
 ],
 "requiresUserAction": true|false
```

}

In this optimized version, the impact field is now an array of objects, each containing the impact type and severity. This makes it more flexible and concise, as it eliminates the need to list every impact type with a severity value, even if it is not applicable. Instead, only the relevant impact types and their severities will be included in the array.

Updated version feeding off an example from Microsoft Updates:

*Please note the “ProductCategory” section as this is what Microsoft currently offers for filters. The idea per the below categories, is that there may be a potential for all cloud vendors to agree on some, or all the categories to be used within one common JSON file.


```
{
  "Provider": "Microsoft",
  "Service": "ServiceName",
  "ProductCategory": "Popular|AI and Machine Learning|Analytics|Blockchain|Compute|Containers|Databases|Developer Tools|DevOps|Hybrid and Multicloud|Identity|Integration|Internet of Things|Management|Media|Migration|Mixed Reality|Mobile|Networking|Security|Storage|Virtual Desktop Infrastructure|Web",
  "UpdateType": "Compliance|Features|Gallery|Management|Microsoft Build|Microsoft Connect|Microsoft Ignite|Microsoft Inspire|Open Source|Operating System|Pricing & Offerings|Regions & Datacenters|Retirements|SDK & Tools|Security|Services",
  "Impact": {
        "Performance": "Low|Medium|High|N/A",
        "Security": "Low|Medium|High|N/A",
        "Availability": "Low|Medium|High|N/A",
        "Compliance": "Low|Medium|High|N/A",
        "Cost": "Low|Medium|High|N/A",
        "UserExperience": "Low|Medium|High|N/A"
  },
  "Description": "Description of the change and its implications",
  "URL": "https://release-notes-url",
```

```
 "EffectiveDate": "2023-04-04T00:00:00",
 "AffectedSettings": [
        {
        "SettingName": "Setting1",
        "PreviousValue": "OldValue",
        "NewValue": "NewValue"
        },
        {
        "SettingName": "Setting2",
        "PreviousValue": "OldValue",
        "NewValue": "NewValue"
        }
 ],
 "RequiresUserAction": true|false
}
```

Change Management Lessons Learned so far:

1. A manual process is needed to map the updates to the JSON schema to get accurate results. It is possible to train a machine learning model, but there would still be a form of formal data review to ensure accuracy.

2. If a web page is where we are feeding updates from, the code base would need to be altered per vendor to ensure that updates are not stalled and remain continuous. The largest roadblock so far is running into HTML layout errors.

3. A common JSON schema agreement would be ideal. However, Azure's product types and categories are quite granular, and could potentially be used cross cloud into other vendors upon a formal agreement.

a. Otherwise, each cloud vendor would need to have a custom JSON script written to match the publishing of their patch notes, updates, and RSS email feeds.

We would need to create a JSON file with vendor specific containers. So, if azure, it can pull from its list of product categories of updates. If AWS, it then autofills the categories for it. This of course is where an agreed upon JSON format would fit into the grand scheme of creation. Each website has a specific manner in which their updates are filtered, categorized, and ordered. But by adopting a general consensus API call (much like an RSS feed would work) this process can be streamlined into a management system.

Additional considerations that have not been researched yet:

- How to map to a company's current control posture to implement a true impact level

  - We can however keep the impacts stagnant to give an overall look of the areas and match that to a business model.

- Control mappings for standards and frameworks.

- Does this need to only be high level? Or do we want to achieve a lower level of technical feasibility. This would require more backend work to tie into systems.