Journal of Technological Innovations

Est. 2020

Cost-Optimizing Infrastructure Platform with Dynamic Code Execution

Naveen Koka

Email: na.koka@outlook.com

Abstract:

A comprehensive system has been established comprising a manager and executor running on a default small instance. The system supports various types of invocations, such as REST API and database updates, each delegating responsibility to the executor. The manager's interface allows the definition of API attributes, including endpoint, HTTP method type, authentication methods (basic, OAuth, username, password), script editor, and the name of dynamic scripts. The code execution environment prioritizes efficiency by default, running 10 threads concurrently on a single small instance.

Upon invocation, system logs are meticulously generated, capturing essential information for error identification and analysis. Additionally, script logs containing relevant data from the code are extracted, saved, and made accessible to users. The system maintains these script logs for a duration of 10 days, facilitating historical analysis.

To optimize resource allocation, the system dynamically manages the instantiation of new instances based on thread utilization. Specifically, if the number of concurrent thread calls surpasses a predefined threshold and there are a significant number of pending requests, a new machine is spawned to handle the additional workload.

Furthermore, version control is integrated seamlessly using GitHub, offering inherent versioning advantages. The code files bear the .gvy extension, while metadata files utilize the .md extension. The manager is equipped to validate scripts, providing compilation errors in a user-friendly displayable format.

In terms of data persistence, the system database stores essential information about APIs and scripts, including global variables identified by {!}. The manager defaults to applying linting and executing csfixer to address any potential code smells. Additionally, the UI allows users to configure various attributes, fostering a user-friendly environment for script definition and customization.

Overall, this abstract encapsulates a versatile and efficient code execution system with robust logging, version control, and resource management capabilities.

Keywords: Dynamic code execution, literals, infrastructure optimization,

Runtime code execution, Parsing code

1. Introduction

In this integrated code execution system, a manager and executor work cohesively to handle diverse invocation types, offering a user-friendly interface for configuring API attributes and global variables. Prioritizing efficiency, the system operates on a default small instance, dynamically managing resources by spawning new instances when necessary.

2. Problem Statement

The widespread availability of cloud services has empowered developers to leverage instances and app services effortlessly. However, this convenience has led to a prevalent practice of keeping instances continuously active, incurring unnecessary costs. Developers often overlook the economic implications of maintaining machines in an active state, resulting in increased expenditure on cloud resources. Therefore, there is a growing need for awareness and optimization strategies to ensure judicious utilization of cloud services and cost-effective code execution practices.

In the absence of comprehensive documentation, organizations often prioritize spending on cloud services, if these services fulfill their intended purposes. The willingness to invest without thorough documentation highlights the perceived value and necessity of the cloud resources in meeting organizational goals. However, emphasizing the importance of documentation can lead to more informed decision-making and efficient resource allocation, ultimately contributing to cost-effective and streamlined operations.

Over time, this approach can turn into a nightmare, leading to a significant escalation in costs that may far exceed the revenue being generated. The lack of strategic resource management and documentation can result in financial challenges, prompting organizations to reassess their cloud spending practices maintaining a sustainable balance between costs and revenue.

3. Solution

We will be creating a light weight engine which will be used to fetch and execute the code in runtime. This way a small instance can be used to execute multiple services at one go.

This also provides the developers to concentrate on writing the code rather than infrastructure.

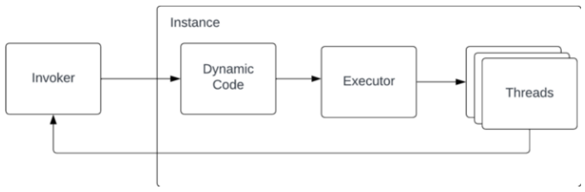


Fig 1: Infrastructure Optimization

Several factors contribute to the realization of infrastructure optimization, and we will delve into each of them in detail.

The Dynamic Code

The code integrated into the executor has a specific function, solely designed to receive data, make modifications, and then return the modified data to the invoker.

The code is authored in Groovy, yet the approach is not confined to a specific language; we can employ anydynamically executable code. To this paper, Groovy will serve as the reference dynamic code base.

While crafting dynamic code, several default libraries are available for reference. Examples include groovy.lang, groovy.util, groovy.json, groovy.sql, groovy.transform, among others. Groovy boasts a comprehensive collection of seamlessly integrated libraries, enhancing its versatility.

During the initial implementation, we will reference methods from the same class.

The Executor

The executor serves as the core engine responsible for interpreting and executing the code within the execution process. It initiates threads, transfers the necessary data and code for execution, and ultimately returns the resulting data.

The code must be pulled from the cache, if it's not available then check with Dynamic code by passing the system record id from the system database.

The executor is crafted in Java to seamlessly integrate with Groovy scripts, benefiting from the compatibility between Groovy and Java. These concepts can also be applied and implemented with any other language integrated with dynamic code capabilities.

The executor functions as a continuously running entity, monitoring and handling incoming requests. It receives requests, extracts the dynamic code specified in the request, and then executes the script using the built-in GroovyShell.

The script is to be executed within the thread by providing both the script and the script context.

Upon initiating the thread, the executor will log pertinent details about the thread, including information about the incoming request, the executed script, and the resulting response. This approach allows for comprehensive storage of information related to each request and its corresponding script execution.

In addition to the previously mentioned details, supplementary information such as the time taken to complete the script, any generated warnings, and the frequency of script usage will also be stored. This comprehensive approach ensures a thorough record of various aspects related to script execution for further analysis and optimization.

The Thread

The thread is the operational space where dynamic code is executed, and tasks are performed. Upon initiating the thread, a small scheduler component is activated to monitor and track its progress.

The guiding principle is that no code should run for more than 60 seconds in a typical transactional system. In instances where code execution exceeds this designated duration, the following steps will be taken.

Each thread is allotted a time limit of 60 seconds for code completion. If there is a need to extend this limit beyond 60 seconds, a new machine is dynamically spawned to execute the code exclusively, without impacting other threads.

Careful consideration of this configuration is essential, as it plays a critical role in minimizing additional costs. Establishing rules around this configuration is prudent, and approval from designated personnel responsible for managing these aspects can be implemented as an additional layer of oversight.

The code parser

If the code operates within the database context, it can create, select, or update the data in the database. To mitigate security-related concerns, the dynamic code will be tasked with translating the OData query language into the corresponding SQL query.

We will parse the file, extract all querys, and replace them with the actual SQL queries. The queries to be identified will be enclosed within {!}. The data will be parsed and returned as a list of JSON data.

To insert the data, Create an object and convert it into a list. To recognize the code, it will be enclosed within {!}, containing two parameters: the variable data and the table name. Using this information, generate the insert command for inserting into the code.

Additionally, we can generate predefined values and utilize them as literals, such as "today," "yesterday," and other date-related operations, etc.

All database operations will be performed in transaction mode to prevent any partial data commits. These transaction statements also need to be included as part of the source code.

We intend to cache this information since parsing the file incurs a significant cost. If there are modifications to the file or it is no longer frequently used, it will be unloaded, while others will remain cached.

The code versioning

The code will be stored on GitHub, with the significant advantage of inherent versioning. Only The Manager has the authority to manage this repository, and the source of edits will exclusively originate from The Manager. To commit and pull the code, interactions with the GitHub APIs are necessary.

Whenever code is committed directly to the main branch, the metadata file must be internally managed, including the recording of the username for the commit.

Administrators must ensure to prevent rate limit errors, and one way to achieve this is by submitting requests to increase the limit.

Proper authentication implementation is essential to prevent any Not Found errors.

Certain errors need to be interpreted to present the user with the appropriate messages. Here are some examples:

Code Out Of Date:

If a user is editing the code concurrently with another user, and user1 has already committed the code, when user2 attempts to commit, a message indicating that the code is out of date should be conveyed. GitHub inherently provides this functionality, and we need to present it to the user in a clear and organized manner.

File Exists:

Avoid duplicating files with the same name if they already exist in the repository.

The Manager

This UI is employed to define the executor and can be customized for invocation in various forms. The manager is responsible for offering a UI to configure the script. The script needs to incorporate a method named "run" that takes a JSON as its parameter, representing the script's body. Certain

metadata, such as file name, created user, and last modified user, is stored as a JSON in a file associated with the code.

· The code file is assigned the extension .gvy.

· The metadata files are given the extension .md.

The manager is responsible for validating the script and returning any compilation errors in a displayable structure.

The manager is required to automatically apply linting and identify and rectify any potential code smells.

Upon meeting all the specified rules, we can proceed to submit the new or modified code to Git for storage. In the event of any errors returned by Git, it is crucial to interpret them gracefully and present the information to the user in a clear and understandable manner.

The metadata file is stored within the document and subsequently submitted to Git. The manager bears the responsibility of maintaining the authenticity of the file, and this information is not disclosed to the user.

The manager possesses a database to store information regarding the API and files to be executed. Global variables, identifiable by {!}, are defined for use in API executions. The UI allows for the specification of the following attributes:

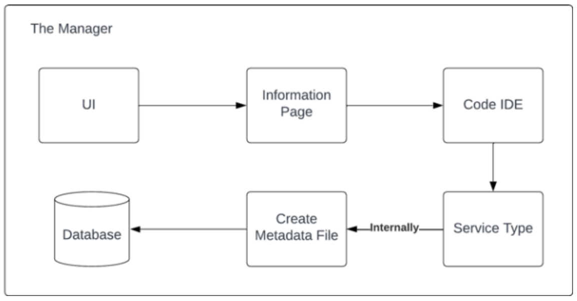
· API endpoint

· HTTP Method Type

· Authentication (supports basic, OAuth, username, and password)

· Script Editor

· Name of the Dynamic Script



3.7 The Invoker Types

The invoker can trigger multiple types and delegate the responsibility to the executor:

1. Rest API:

· Define the REST API and host it internally.

2. DB updates:

· Invoked when any record is inserted or updated. This is also exposed through an API but is primarily driven by changes in the database tables.

The Infrastructure Optimization

A default small instance is continuously running the manager and executor. When the code is invoked by one of the types, a thread becomes available to handle the request without requiring additional infrastructure. By default, 10 threads are concurrently running on one small instance. If the number of concurrent thread calls exceeds 5 and there are 15 or more pending requests, a new machine will be spawned to execute the new set of requests.

The Metrics

At regular intervals, such as every month, we can furnish metrics illustrating the amount saved on infrastructure costs. This reporting mechanism will offer insights into the efficiency gains and cost reductions achieved through optimized infrastructure utilization.

The metrics encompass additional details stored within the thread executor. Tracking can be performed based on the number of lines executed, providing a granular measure of script performance and resource utilization. The metrics encompass additional details stored within the thread executor. Tracking can be performed based on the number of lines executed, providing a granular measure of script performance and resource utilization.

Additionally, we can improve the system to capture details regarding code execution and other relevant information to enrich the metrics.

The Logs

System Logs:

All logs generated during a transaction will be saved. This aids in identifying any encountered errors.

Script Logs:

Logs created within the code will be extracted, saved, and presented to the user. This historical log data will be stored for the past 10 days.

The manager should feature a visual representation to display log data, offering enhanced insights for both administrators and users regarding the ongoing execution. This not only provides a modern touch but also facilitates a step-by-step understanding of the execution process.

4. Technical Considerations

Management Component

UI, web server, and backend code for visual interaction.

Utilizes React for the frontend.

Web server options include Tomcat or Node.js.

Backend implemented in Java.

Executor Component

Implemented in Java and exposed as a REST API.

Database

Utilizes PostgreSQL to store all configurations. Utilize the same database for the logs storage

Document Storage

GitHub serves as the document storage system.

5. Uses

Utilizing infrastructure optimization through the execution of dynamic code offers several benefits:

Cost Efficiency

Maximizes resource utilization, minimizing unnecessary costs associated with idle or underutilized infrastructure.

Performance Enhancement

Optimizes the execution environment, leading to improved performance and reduced execution times for code.

Scalability

Enables dynamic scaling by spawning new instances based on demand, ensuring efficient resource allocation during peak workloads.

Version Control Integration

Integrates seamlessly with version control systems like GitHub, providing versioning benefits and enhancing code management practices.

Cost Tracking and Analysis

Allows organizations to track and analyze costs associated with infrastructure usage, facilitating informed decision-making and resource allocation.

Flexibility and Adaptability

Facilitates easy adaptation to changing workloads and requirements, ensuring the infrastructure aligns with the dynamic nature of development and operations.

Resource Monitoring

The system is engineered to furnish organizations with versatile infrastructure management and code execution, ensuring comparable performance with reduced costs.

6. Use Cases

Banking Sector

By using the infrastructure there is a direct benefit of the cost savings as well as the security can be implemented to make sure the dynamic code can be used.

Electronic Manufacturers

There are so many tools needed to quickly develop and deploy with in no time. This concept can be used and develop and deploy in no time.

7. Conclusion

In conclusion, failing to proactively manage our infrastructure can result in escalating costs. To address this, we need a system that dynamically regulates infrastructure usage, optimizing resources to deliver desired services efficiently. This system introduces a distinctive approach to code management and execution, reducing dependency on exhaustive documentation for every piece of code.

8. References

[1] Vlad Nevzorov. (2011). How much does it cost to develop a line of code?. https://vladnevzorov.wordpress.com/2011/01/31/how-much-does-it-cost-to-develop-a-line-of-code-sloc/

[2] shepelev. 2021. Why Git Is A Great Documentation Management Tool. https://hackernoon.com/why-git-is-a-great-documentation-management-tool-p712339s

[3] Xianglong Huang, Brian T Lewis, Kathryn S McKinley. 2006. Dynamic code management: improving whole program code locality in managed runtimes. https://dl.acm.org/doi/abs/10.1145/1134760.1134779.

[4] HERB KRASNER. (2021). The Cost of Poor Software Quality in the US: A 2020 Report. https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf.

[5] Sarah Wang, Martin Casado. (2021). The Cost of Cloud, a Trillion Dollar Paradox. https://a16z.com/the-cost-of-cloud-a-trillion-dollar-paradox/

[6] Jorge Manrubia (2009). Evaluating code dynamically in Groovy (differences with Ruby). https://www.jorgemanrubia.com/2009/10/10/evaluating-code-dynamically-in-groovy/

[7] Cheney Shue (2015). Execute Groovy dynamically. https://forums.oracle.com/ords/apexds/post/execute-groovy-dynamically-3067