University of Cyprus
Department of Computer Science

Exploring Swift as a Serverless Language on OpenWhisk

Andreas Loizides

Mentor: Dr Haris Volos

May 2023

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Chapter 1. Introduction

1.1. Background

1.2. Research Questions

1.2.1 Performance Comparison

How does the performance of Swift compare to other serverless languages, such as Python, Java, and Node.js? We can run synthetic benchmarking workloads on OpenWhisk to compare the performance of Swift with other languages.

1.2.2 Cold Start Analysis

Cold start refers to the time taken to initialize a new serverless function. How does the cold start time of Swift compare with other serverless languages? We can use OpenWhisk to run tests to analyze the cold start time of Swift.

1.2.3 Ease of Development

How easy is it to develop serverless functions in Swift compared to other languages? We can evaluate the development experience of Swift by comparing it with other serverless languages.

1.2.4 Resource Utilization

How efficiently does Swift utilize serverless resources, such as CPU and memory? You can use OpenWhisk to monitor resource utilization and analyze the efficiency of Swift in utilizing these resources.

1.2.5 Cost Analysis

How does the cost of running serverless functions in Swift compare with other languages? We can compare the cost of running Swift functions on OpenWhisk with the cost of running functions in other serverless languages.

Chapter 2. Ease of Development

How easy is it to develop serverless functions in Swift compared to other languages? We can evaluate the development experience of Swift by comparing it with other serverless languages.

2.1. Language Simplicity and Syntax

Swift is a modern, expressive, and safe programming language designed for performance and ease of use. Its simplicity and syntax make it a strong candidate for serverless functions. In this section, we will examine how Swift's syntax compares to other languages and how it contributes to a better development experience.

Simplicity in Swift

Swift's syntax aims to be concise and clear, which can lead to shorter and more readable code. For instance, consider the following example of declaring a constant in Swift:

```
let numberOfDays = 7
```

This is similar to Python:

```
number_of_days = 7
```

However, in Java, the declaration is more verbose:

```
final int numberOfDays = 7;
```

Swift's simplicity becomes more evident when dealing with more complex constructs, such as optional values. In Swift, optional values are handled using the ‘¿ and ‘¡ operators, making it easy to declare and unwrap optional values:

```
let optionalValue: Int? = 42
let unwrappedValue: Int = optionalValue!
```

In comparison, Java uses ‘Optional‘ containers, which leads to more verbose code:

```
Optional<Integer> optionalValue = Optional.of(42);
int unwrappedValue = optionalValue.get();
```

Safety Features

Swift’s syntax includes features that promote safe programming practices and reduce the likelihood of errors. One such feature is the guard clause. The guard statement allows developers to perform early exits from a function or loop, simplifying the code and making it more readable. With the compiler enforcing early exits, developers are less likely to introduce errors.

Here’s an example of a guard clause in Swift:

```
func processInput(_ input: String?) {
    guard let unwrappedInput = input else {
        print("Invalid input")
        return
    }

    // Continue processing with unwrappedInput
}
```

A similar function in Python might use a conditional statement:

```
def process_input(input: Optional[str]):
    if input is None:
        print("Invalid input")
        return

    # Continue processing with input
```

While both versions are readable, the Swift version explicitly enforces the early exit, making it less prone to errors.

Readability

Swift’s syntax and features contribute to more readable code. For example, Swift’s type inference system allows developers to write cleaner code without explicitly declaring types:

```swift
let message = "Hello, world!"
```

In contrast, Java requires explicit type declarations:

```java
String message = "Hello, world!";
```

Additionally, Swift's support for functional programming constructs, such as map, filter, and reduce, can make code more readable and expressive. Here's an example of filtering and transforming an array in Swift:

```swift
let numbers = [1, 2, 3, 4, 5]
let evenSquares = numbers.filter { $0 % 2 == 0 }.map { $0 * $0 }
```

A similar operation in Java is more verbose and less expressive:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenSquares = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
```

Copy-on-Write (CoW)

Copy-on-write (CoW) is an optimization technique that Swift uses to minimize memory usage and improve performance when working with value types like arrays, dictionaries, and strings. CoW delays the copying of a value until it is modified, reducing unnecessary copying and memory overhead.

Here's an example of CoW in action:

```swift
var array1 = [1, 2, 3]
var array2 = array1
array2.append(4)
```

In this example, ‘array1‘ and ‘array2‘ initially share the same underlying storage. When ‘array2.append(4)‘ is called, Swift detects that the storage is shared and creates a separate copy of the storage for ‘array2‘ before appending the value.

A similar example in Java does not benefit from the CoW optimization:

```java
List<Integer> list1 = new ArrayList<>(Arrays.asList(1, 2, 3));
List<Integer> list2 = new ArrayList<>(list1);
list2.add(4);
```

In this case, Java creates a new copy of the list immediately, regardless of whether modifications are made, which can lead to higher memory usage and decreased performance.

Extensions

Swift's extensions allow developers to add new functionality to existing types, such as classes, structures, or protocols, without modifying their original implementation. This feature makes it simple to enhance types in a clean and modular way.

For example, consider adding a method to the 'Int' type that checks if a number is even:

```
extension Int {
    var isEven: Bool {
        return self % 2 == 0
    }
}

let number = 42
print(number.isEven) // Output: true
```

Java does not have an equivalent feature, so a utility method or wrapper class would be needed:

```
public static boolean isEven(int number) {
    return number % 2 == 0;
}

int number = 42;
System.out.println(isEven(number)); // Output: true
```

Swift's extension feature leads to more elegant and expressive code compared to Java in this example.

Custom Operators

Swift allows developers to define custom operators, providing flexibility and expressiveness in the language. Custom operators can be created for arithmetic, comparison, logical, and other operations, enhancing readability and simplifying complex expressions.

For example, consider defining a custom '**' operator for exponentiation:

```
infix operator **: MultiplicationPrecedence

func **(base: Double, exponent: Double) -> Double {
```

```
    return pow(base, exponent)
}

let result = 2.0 ** 3.0 // Output: 8.0
```

In Java, you would need to use the ‘Math.pow‘ function directly, which may be less expressive:

```
double result = Math.pow(2.0, 3.0); // Output: 8.0
```

In summary, Swift’s simplicity and syntax, safety features like guard clauses, support for functional programming constructs, copy-on-write optimization, extensions, and custom operators contribute to writing simple, safe, and efficient code. These features enable developers to create high-performance serverless functions while maintaining readability and expressiveness.

Property Wrappers

Property wrappers in Swift are a powerful language feature that can significantly simplify code by encapsulating common patterns and behaviors. They allow developers to create reusable, composable, and declarative abstractions for property access patterns.

One prominent example of property wrappers is SwiftUI, a user interface toolkit for building applications. SwiftUI extensively uses property wrappers, such as ‘@State‘, ‘@Binding‘, and ‘@ObservedObject‘, to manage state and data flow. These wrappers enable developers to create complex and reactive UIs with concise and expressive code.

Another instance where property wrappers are beneficial is the Swift ‘ArgumentParser‘ library. This library provides a straightforward way to parse command-line arguments. By employing property wrappers like ‘@Option‘, ‘@Argument‘, and ‘@Flag‘, developers can define command-line interfaces with minimal code, eliminating the need to manually handle argument parsing. This results in a more readable and maintainable codebase.

In the following Swift ‘ArgumentParser‘ example, a ‘transform‘ argument is used to define a URL option and validate the input:

```
import ArgumentParser

struct ExampleApp: ParsableCommand {
    @Argument(help: "Your name")
```

```
    var name: String

    @Option(help: "Your age")
    var age: Int

    @Argument(help: "URL of the PS (powersoft) Server", transform: urlTransformer)
    var psURL: URL

    func run() {
        print("Hello, \(name)! You are \(age) years old.")
        print("PS Server URL: \(psURL)")
    }
}

let urlTransformer: (String) -> URL = { str in
    guard let url = URL(string: str) else {
        fatalError("\(str) is not a valid URL!")
    }
    return url
}

ExampleApp.main()
```

For comparison, let's look at similar functionality in Python using the 'argparse' library:

```
import argparse
import sys
from urllib.parse import urlparse

def url_transformer(str):
    url = urlparse(str)
    if not url.scheme or not url.netloc:
        sys.exit(f"{str} is not a valid URL!")
    return url

parser = argparse.ArgumentParser(description="Example command-line application")
parser.add_argument("name", help="Your name")
parser.add_argument("age", type=int, help="Your age")
```

```
parser.add_argument("psURL", type=url_transformer, help="URL of the PS (powersoft

args = parser.parse_args()

print(f"Hello, {args.name}! You are {args.age} years old.")
print(f"PS Server URL: {args.psURL}")
```

And in Java using the ‘picocli‘ library:

```
import java.net.MalformedURLException;
import java.net.URL;
import picocli.CommandLine;
import picocli.CommandLine.Command;
import picocli.CommandLine.Option;
import picocli.CommandLine.Parameters;

@Command(name = "ExampleApp", description = "Example command-line application")
public class ExampleApp implements Runnable {
    @Parameters(index = "0", description = "Your name")
    private String name;

    @Option(names = {"-a", "--age"}, description = "Your age")
    private int age;

    @Parameters(index = "1", description = "URL of the PS (powersoft) Server", co
    private URL psURL;

    public static void main(String[] args) {
        int exitCode = new CommandLine(new ExampleApp()).execute(args);
        System.exit(exitCode);
    }

    @Override
    public void run() {
        System.out.printf("Hello, %s! You are %d years old.%n", name, age);
        System.out.printf("PS Server URL: %s%n", psURL);
    }

    public static class UrlConverter implements CommandLine.ITypeConverter<URL> {
```

```
        @Override
        public URL convert(String value) throws MalformedURLException {
            URL url = new URL(value);
            if (url.getProtocol() == null || url.getHost() == null) {
                throw new MalformedURLException(value + " is not a valid URL!");
            }
            return url;
        }
    }
}
```

In the Python and Java examples, the code for validating and parsing the URL input is more verbose compared to the Swift version. The Swift @Argument property wrapper, combined with the transform argument, simplifies the code and enhances expressiveness.

In summary, Swift's property wrappers contribute to the language's simplicity and expressiveness, making it an attractive choice for serverless development. Their ability to provide powerful abstractions and code simplification is an advantage that might be more difficult or impossible to achieve in other languages.

2.2. Available Libraries and Frameworks

Swift has a growing ecosystem that is constantly evolving and expanding. However, it might not be as extensive as more established languages like Python, Java, and Node.js. This could affect the availability of libraries and frameworks needed for specific use cases in the context of FaaS. In this section, we will discuss some of the notable libraries and frameworks available for Swift, as well as some of the challenges developers may face in certain scenarios.

2.2.1 Web Development

One of the most prominent Swift web frameworks is Vapor, which allows developers to build web applications and APIs using Swift. While Vapor has gained popularity and offers many useful features, it comes with its own set of challenges when compared to popular frameworks or solutions available for Python, Node.js, and Java.

Popular Frameworks for Python, Node.js, and Java

In contrast to Swift's Vapor, more established languages such as Python, Node.js, and Java offer a wide range of popular frameworks and libraries that make web development easier and faster. Some of these frameworks have been around for many years and have extensive documentation, community support, and a large ecosystem of plugins and extensions.

Python

- **Django**: A high-level web framework that promotes rapid development and clean, pragmatic design. It includes an ORM, authentication support, an admin interface, and many other features out-of-the-box [1].

- **Flask**: A lightweight web framework that provides flexibility for developers to choose their own components, such as databases and authentication systems [2].

Node.js

- **Express**: A minimal and flexible web application framework for Node.js that provides a robust set of features for web and mobile applications. Express is widely used and has a large number of plugins and middleware available [3].

- **Koa**: A next-generation web framework for Node.js, created by the team behind Express. Koa is designed to be more expressive and robust while being smaller and more lightweight [4].

Java

- **Spring Boot**: A widely-used framework that simplifies the development and deployment of Java-based web applications. Spring Boot provides built-in support for embedded servers, security, data access, and more [5].

- **JavaServer Faces (JSF)**: A Java web application framework that simplifies building user interfaces for Java EE applications. JSF provides a component-based approach, allowing developers to build UIs by assembling pre-built components [6].

Example Use Cases

1. **User Authentication and Authorization**: When building a web application, it is common to require user authentication and authorization. Django includes built-in support for user authentication, while Express has the popular Passport middleware, and Spring Boot provides Spring Security. On the other hand, Vapor offers authentication support, but it might not be as mature or feature-rich as these other frameworks.

2. **Database Integration**: Many web applications require integration with databases. Django comes with a built-in ORM, while Flask has SQLAlchemy and Spring Boot offers JPA and Hibernate. In the Node.js ecosystem, Sequelize and TypeORM are popular choices for database integration. Vapor has its own ORM called Fluent, but it may lack the maturity, community support, and extensive documentation compared to the other solutions.

3. **Template Engines**: Rendering server-side templates is a common task in web development. Python's Django and Flask both offer built-in template engines, while Node.js's Express supports various templating engines like Pug and EJS. In Java, Thymeleaf is a popular template engine for Spring Boot applications. Vapor supports the Leaf template engine, but it may not have the same level of community support or plugin ecosystem as the alternatives.

In summary, while Swift's Vapor framework offers a powerful solution for web development, it may face challenges in terms of maturity, community support, and available plugins when compared to the popular frameworks and libraries available for Python, Node.js, and Java. Developers should consider the specific requirements of their serverless web applications and weigh the trade-offs between the available options.

Vapor and Swift Package Manager

Due to the way the Swift compiler and the Swift Package Manager currently work, resolving package dependencies can be time-consuming, as it happens sequentially. Even a boilerplate Vapor app requires the compilation of more than 1,500 files. Additionally, the Swift Package Manager has experienced stability issues, such as a bug related to updating dependencies from a specific branch of a Git repository. Although the bug has now been resolved, it demonstrates the growing pains Swift's ecosystem is experiencing.

Documentation and Community Support

As a newer language, Swift's documentation and community support might not be as comprehensive as more established languages. Developers may face challenges in finding appropriate resources or examples when working with specific libraries or frameworks in the context of FaaS. This could lead to slower development and increased reliance on trial and error to find solutions.

Despite these challenges, Swift's ecosystem continues to grow and improve, and many developers are contributing to its progress. With time and continued investment from the community, Swift is likely to become a more mature and stable language for serverless functions.

Example: Swift Package Manager Bug

One example that highlights Swift's infancy and lack of maturity for production use is a bug related to the Swift Package Manager. In the past, the Swift Package Manager had an issue where the package cache would not get invalidated for recent commits when a dependency was set to a branch of a Git repository (e.g., `.package(url: "https://github.com/andreas16700/OTModelSyncer_pub", branch: "main"`). This bug significantly hindered development, as developers were forced to resort to obscure workarounds to manage their dependencies.

In the specific case mentioned, the project was divided into several packages, and changes were committed frequently. Due to this bug, the development process became considerably more challenging. The developer had to run unit tests and use Docker containers to ensure operational consistency with Linux, which resulted in increased development time and effort.

Although the bug has now been fixed, it took more than three years since it first appeared on the Swift forums[1]. This example demonstrates that while Swift is a powerful and promising language, it still faces challenges due to its relative immaturity compared to more established languages.

As the Swift ecosystem continues to grow and mature, it is likely that such issues will become less common. However, developers should be aware of the potential challenges they may face when adopting Swift for serverless functions and be prepared to invest additional time and effort to address them.

[1] "Dependency on (very) active branch doesn't pull new commits", Swift Forums, January 2020 https://forums.swift.org/t/dependency-on-very-active-branch-doesnt-pull-new-commits/33204

2.2.2 Developer community and support

A robust developer community and available resources can contribute to the ease of development. While Swift has a strong community, the serverless aspect of it might not be as well-established as other languages.

2.2.3 Tooling and IDE support

Swift has great support in Xcode, which is the primary development environment for Apple platforms. However, developers who primarily work with other languages may not be familiar with Xcode. There is also support for Swift in other IDEs, such as Visual Studio Code, but the level of support may vary compared to languages with more extensive serverless development history.

2.2.4 Integration with serverless platforms

The ease of integrating Swift with serverless platforms like OpenWhisk, AWS Lambda, or Google Cloud Functions may impact the development experience. Consider the availability of templates, plugins, and other tools that facilitate serverless development and deployment.

2.2.5 Learning curve

The ease of development in a language also depends on an individual developer's familiarity and previous experience with that language. Swift is easy to learn for beginners, but developers who have never used Swift may need some time to become proficient.

2.2.6 Linux Support

Swift's support for Linux is relatively recent, and there are some features that are not available or have limited functionality due to the absence of the Objective-C runtime. Swift was designed to interoperate closely with Objective-C when it is present, but it was also designed to work in environments where the Objective-C runtime does not exist[2]. This means that some features that depend on the Objective-C runtime are not available on Linux.

For example, when a Swift class on Apple's platforms is marked `@objc` or subclasses `NSObject`, you can use the Objective-C runtime to enumerate available meth-

[2] "Platform Support", swift.org `https://swift.org/about/#platform-support`

ods on an object or call methods using selectors. Such capabilities are absent on Linux because they depend on the Objective-C runtime[3].

Developers might encounter unexpected limitations when developing serverless functions due to these differences. The exact features that are not available on linux are not well documented. While a program may compile and run fine on a Mac, it may either encounter a compiler-time error on linux or, albeit sparingly, crash at runtime. Developers may need to resort to writing unit tests and running them in Docker containers to ensure operational consistency between platforms.

An Example

While developing a crucial serverless function, compiling and deploying it raised no concerns. But trying to invoke it elicited an unexpected error.

```
aloizi04@kube1:~$ python3 invoke.py -p 8083 init syncModel syncModel_action.zip
{"ok":true}

aloizi04@kube1:~$ python3 invoke.py -p 8083 run
{"error":"command exited"}
```

Figure 2.1: Invoking the action using Apache's invoke.py tool

Note: This is using the *invoke tool*[4] provided by OpenWhisk for debugging runtimes.

Every action runs on a language runtime. A runtime is a docker image. Attaching to the docker container and trying to run the binary to get more clues, we can see the following issue:

```
root@d0478e2b14c9:/swiftAction/action/1/bin# ./exec
{"value": {}}
Received: {"value": {}}
Syncing model965
fetching source PS data..
Illegal instruction (core dumped)
root@d0478e2b14c9:/swiftAction/action/1/bin#
```

Figure 2.2: Crash witnessed by manually running the action executable in the runtime container

Due to the lack of Swift Linux IDEs and debuggers, not many options besides using `lldb` exist.

Building the program again in debug mode (for easier debugging) and investigating with `lldb`, an offending function is found and a breakpoint is set.

The issue seems to arise after invoking the `dataTask(with:)` function of Foundation's `URLSession`. The networking part of `Foundation` made heavy use of

[3]ibid

[4]`https://github.com/apache/openwhisk/blob/master/tools/actionProxy/invoke.py`

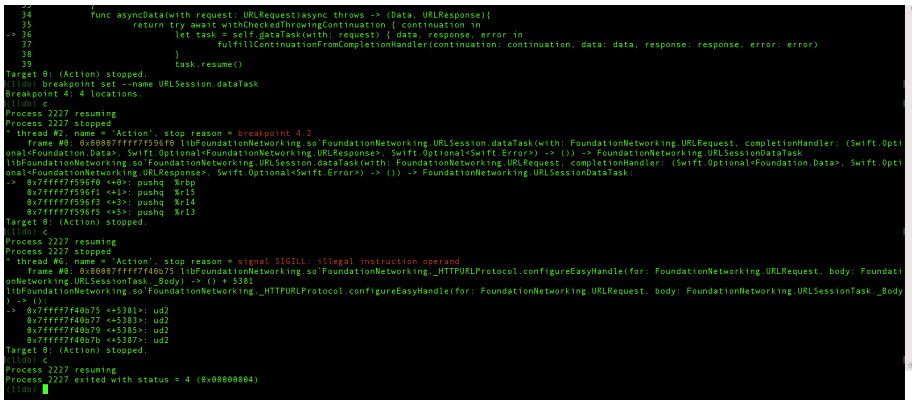

Figure 2.3: Debugging with LLDB to find the cause of the crash

Objective-C and the Swift team, in the transition away from Objective-C, made a separate package called `FoundationNetworking` just for Linux. Many networking parts of `Foundation` are either poorly documented about their Linux support or silently unimplemented. The true cause of the above crash is unknown.

This example demonstrates a significant hurdle in the development process that might arise with no compiler warning, and made exceptionally more difficult with the sheer lack of any sort of feedback.

Debugging serverless applications is already difficult enough, and Swift FaaS developers should expect it to be even more difficult (in the case of Swift) as they may find themselves using `lldb` unexpectedly.

Swift's Linux support continues to evolve, and the community is working on addressing these limitations. However, developers should be aware of the potential challenges they may face when adopting Swift for serverless functions in a Linux environment and be prepared to invest additional time and effort to overcome them.

Bibliography

[1] D. Project: "Django," nd, https://www.djangoproject.com/.

[2] Pallets: "Flask," nd, https://flask.palletsprojects.com/.

[3] Express.js: "Express," nd, https://expressjs.com/.

[4] K. Team: "Koa," nd, https://koajs.com/.

[5] P. Software: "Spring boot," nd, https://spring.io/projects/spring-boot.

[6] O. Corporation: "Javaserver faces," nd, https://www.oracle.com/java/technologies/javaserverfaces.html.

Chapter A. Appendix A

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.