

Tech tornadoes team

Hackathon (“Hello world”)

Prerequisites

- ChatGPT v4 (GPT Omni v4)
- Project requirements and initial architecture concept
- Web client for sending requests to the ChatGPT backend

Prompts for microservice choose

1. Please, choose the best microservice framework for the languages out of the list {PROGRAMING_LANGUAGES_LIST}. The framework will be used for development of a {PROGRAM_NAME}. It means that it is essential to support the features {FEATURES_LIST}.
2. Also keep in mind, that future {PROGRAM_NAME} will be implemented as a set of microservices. It is essential to build efficient asynchronous interactions between them.
3. There will be {NUMBER_OF_MICROSERVICES} microservices: {MICROSERVICES_LIST}.
4. The framework must support {API_DESIGN_PARADIGMA}.
5. The framework must have built-in support for {MECHANISM}.
6. The solution must be {CONSTRAINTS_LIST}.

Initial criterias for the first prompt: languages

Please, choose the best microservice framework for the languages out of the list:

Polyglot

C

C++

C#

D

Erlang VM

Go

Haskell

Java VM

Node.js

Perl

PHP

Python

Ruby

Rust

Initial criterias for the first prompt: project requirements

The framework will be used for development of a load blancer with a service turn-off. It means that it is essential to support the features:

Zero Scaling

Allow applications to scale down to zero instances when not in use and automatically scale up when required.

Event-Driven Scaling

Use complex event processing to trigger scaling actions based on defined monitoring metrics and thresholds.

Custom Resource Allocation

Tailor the allocation of compute resources specifically for ML applications, ensuring they get the necessary resources when scaling up.

Integration with Monitoring Tools

Seamlessly integrate with existing monitoring tools to gather real-time data for making scaling decisions.

Dashboards and Alerts

Provide user-friendly dashboards to visualize events and performance metrics, and set up alerts for critical events.

Automata-Based Policy Engine

Deploy a sophisticated policy engine employing complex automata for precise and customizable scaling rules.

API for Customization

Offer an API that allows for customization and tuning of the scaling algorithms to fit specific application needs.

Initial criterias for the first prompt: constraints

- Also keep in mind, that future load balancer will be implemented as a set of microservices. It is essential to build efficient asynchronous interactions between them.
- There will be 4 microservices: for load balancer, balancing policies controller, k8s node manager and prometheus metrics reporter
- The framework must support RESTful API creation
- The framework must have built-in support for pub/sub mechanism of reaction on asynchronous events.
- The solution must be scalable and handle high (more than 25000) RPS rate

Results of comparison:

Go with Gin/Gonic

OR

Rust with Actix Web

Prompt for the Go solution:

Use Go with Gin/Gonic to create a simple program that returns HTTP response with "Hello world" on a [/ping](#) route

Please, provide instruction how to install go and gin with brew on MacOS

Code for the Go solution:

Pros:

- Performance: Go is designed to handle high concurrency with its lightweight goroutines, making it ideal for high RPS scenarios.
- Low Memory Footprint: It's compiled to machine code, which reduces latency and resource usage compared to interpreted languages.
- Gin Framework: Gin is a lightweight HTTP web framework that provides very high performance and minimal overhead for RESTful services.
- Built-in Concurrency: Go's concurrency model with goroutines allows effective utilization of CPU and memory, handling many simultaneous connections.
- Microservices-Friendly: Go has robust support for building microservices and integrating with container orchestration systems like Kubernetes.

Cons:

- Younger Ecosystem: Compared to Java, Go has a less mature ecosystem, although it's growing rapidly.
- More Boilerplate Code: Go tends to require more manual handling of certain aspects such as dependency injection.

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
    // Create a new Gin router
    instance
    router := gin.Default()

    // Define a GET route for /ping
    router.GET("/ping", func(c
    *gin.Context) {
        // Send a JSON response
        with "message": "Hello world"
        c.JSON(200, gin.H{
            "message": "Hello
world",
        })
    })

    // Run the server on default
    port 8080
    router.Run(":8080")
}
```

Rust with Actix Web: prompt

Prompt: Generate hello world microservice using {FRAMEWORK_NAME} which will receive request by HTTP

Pros:

- **High Performance:** Rust is known for its performance, zero-cost abstractions, and safety, making it well-suited for systems requiring high RPS.
- **Memory Safety:** Rust ensures memory safety without a garbage collector, which helps prevent common bugs and results in predictable performance.
- **Actix Web:** It's one of the fastest web frameworks, optimized for performance, with excellent support for asynchronous programming using Rust's async features.
- **Concurrency:** Leveraging Rust's async capabilities and Actix's actor model allows handling many connections efficiently.
- **Safety and Speed:** Rust's focus on safety and speed through its ownership model results in fewer runtime errors and optimal performance.

Cons:

- **Complex Learning Curve:** Rust's system-level control introduces a more complex learning curve compared to languages like JavaScript or Python.
- **Ecosystem Maturity:** Although rapidly evolving, some parts of the Rust ecosystem may not be as mature as Java's.

Rust with Actix Web: result

Below is an example of a simple "Hello, World!" microservice using Rust and Actix Web that can handle HTTP requests:

Cargo.toml

First, create a new Rust project (e.g., using `cargo new hello_world_microservice`). Then, edit the `Cargo.toml` file to include the Actix Web dependency:

toml

 Копировать код

```
[package]
name = "hello_world_microservice"
version = "0.1.0"
edition = "2021"

[dependencies]
actix-web = "4.0"
```

Rust with Actix Web: result

src/main.rs

Next, create the service in the `src/main.rs` file:

```
rust Копировать код

use actix_web::{web, App, HttpServer, Responder, HttpResponse};

// Handler function for the root path
async fn hello_world() -> impl Responder {
    HttpResponse::Ok().body("Hello, World!")
}

#[actix_web::main]
async fn main() -> std::io::Result<> {
    // Start HTTP server
    HttpServer::new(|| {
        App::new()
            // Define route and attach the handler
            .route("/", web::get().to(hello_world))
    })
    .bind("127.0.0.1:8080")? // Bind server to localhost:8080
    .run()
    .await
}
```

Rust with Actix Web: result

Run the Microservice

1. Build and run the project:

```
bash
```

 Копировать код

```
cargo run
```

2. The server will start on `http://127.0.0.1:8080` . Open your browser or use a tool like `curl` to send an HTTP request:

```
bash
```

 Копировать код

```
curl http://127.0.0.1:8080
```

3. You should see the response:

```
Hello, World!
```

 Копировать код

This is a basic example. You can extend it by adding more routes, middleware, and integrations like databases or logging as required.