

# gem5 Resources



# What are Resources? (Disks, kernels, binaries, etc.)

- gem5 resources are prebuilt artifacts that can be used to run gem5 simulations.
- Each gem5 resource falls into one of 13 categories (such as binary or kernel) and supports one of 6 ISAs (including ARM, x86, and RISC-V).
- For more information about categories, visit [resources.gem5.org/category](https://resources.gem5.org/category).
- The [gem5 resources website](https://resources.gem5.org) is an easy way to search for the resources you want to use.
  - There are filters based on category, ISA, and gem5 version that help you narrow down the resources based on your requirements.



# Important categories and their description

**Kernel:** A computer program that acts as the core of an operating system by managing system resources.

**disk-image:** A file that contains an exact copy of the data stored on a storage device.

**binary:** A program that is used to test the performance of a computer system.

**bootloader:** A small program that is responsible for loading the operating system into memory when a computer starts up.

**checkpoint:** A snapshot of a simulation.

**simpoint:** This resource stores all information required to perform a Simpoint creation and restore.

**file:** A resource consisting of a single file.

**workload:** Bundles of resources and any input parameters that can be run directly in gem5.

**suite:** A collection of workloads.

# Resource Versioning

- In gem5, all resources have an `id` and any update to the resource will update the `resource_version`.
- Each unique resource is represented by its `id` and `resource_version`.
- When an existing resource is updated the `id` remains the same but the `resource_version` is updated.
- Each resource also has a field called `gem5_versions` which shows which releases of gem5 the resource is compatible with.



# Using Resources in gem5 Simulations

To use the resources in gem5, we can use the `obtain_resource` function.

Lets do an example to use the `x86-hello64-static` binary in an example.

Go to the [materials/02-Using-gem5/02-gem5-resources/01-hello-example.py](https://github.com/gem5/gem5/blob/master/materials/02-Using-gem5/02-gem5-resources/01-hello-example.py).

This file builds a basic board and we use the `x86-hello64-static` resource and run the simulation.



# Run the hello binary

To get the binary we write the line

```
board.set_se_binary_workload(observe_resource("x86-hello64-static"))
```

Lets break down this code

- The part `observe_resource("x86-hello64-static")` gets the binary from gem5 resources (go into detail about the parameters)
- The part `board.set_se_binary_workload` tells the board to run the binary that it is given.

Then we run the simulation

```
gem5 01-hello-example.py
```

# Workloads

Workload is a package of one or more resources that can have pre-defined parameters.

lets see the `x86-npb-is-size-s-run` workload.

This workload runs the npb is benchmark in SE mode.

You can see the JSON of the workloads in the [raw](#) tab on the resources website.

```
{
  "category": "workload",
  "id": "x86-npb-is-size-s-run",
  "description": "This workload run the npb-is-size-s binary in SE mode.",
  "function": "set_se_binary_workload",
  "resources": {
    "binary": {
      "id": "x86-npb-is-size-s",
      "resource_version": "1.0.0"
    }
  },
  "architecture": "X86",
  "tags": [
    "npb",
    "x86"
  ],
  "code_examples": [],
  "license": "NASA Open Source Agreement (NOSA)",
  "author": [
    "NASA"
  ],
  "source_url": "",
  "resource_version": "1.0.0",
  "gem5_versions": [
    "23.1",
    "24.0"
  ],
  "example_usage": "obtain_resource(\"x86-npb-is-size-s-run\")",
  "additional_params": {}
}
```

## Workloads (Conti.)

Lets see the `x86-ubuntu-24.04-boot-with-systemd` workload, you can see the [raw](#) tab to see how the resource is made.

- The `function` field has the name of the function that the workload calls.
- The `resources` field contains the resources that the workload uses.
  - The key of the `resources` field like `kernel`, `disk_image`, etc are named the same as the parameter name in the function that the workload calls.
- The `additional_params` fields contains values of non-resource parameters that we want the workload to have
  - `kernel_args` parameter we are using in the above workload.



# Suites

Suites are a collection of workloads, that can be run in parallel using multiprocessing (This will be shown later)

All workloads in the suite have something called `input_groups` that can be used to filter the suite.

Lets do an example that used the suite. In this example we will:

- Print all the workloads in the suite
- Filter the suite with `input_groups`
- Run a workload from the suite

# Printing all the workloads in a suite

The `SuiteResource` class act as a generator so we can iterate through the workloads.

Lets print the `id` and `resource_version` of the workloads in the `x86-getting-started-benchmark-suite` suite.

Modify [02-suite-workload-example.py](#)

Let's get the resource and iterate through the suite and print id and version of the workloads

```
getting_started_suite = obtain_resource("x86-getting-started-benchmark-suite")

for workload in getting_started_suite:
    print(f"Workload ID: {workload.get_id()}")
    print(f"workload version: {workload.get_resource_version()}")
    print("=====")
```

## Filtering suites by `input_groups`

Each workload in a suite has some `input_group` that we can filter it by.

Lets print all the unique input groups the suite has

we can do this by using the `get_input_groups()` function

```
print("Input groups in the suite")  
print(getting_started_suite.get_input_groups())
```

# Running a workload from the suite (single code block)

Lets run the npb IS benchmark that is in the suite

We need to filter the suite so that we get the workload we want

lets filter the suite to return a suite that contains all workloads that have the input tag `is`, we convert the returned object to a list and get the first workload in it.

This works because `is` is a unique tag that only one workload has.

lets print the id of our workload

Lets run the workload with the board that we have.

```
npb_is_workload = list(getting_started_suite.with_input_group("is"))[0]
print(f"Workload ID: {npb_is_workload.get_id()}")
board.set_workload(npb_is_workload)
```

## Local resources

You can also use resources that you have created locally in gem5.

You can create a local JSON file that you can use as a data source by setting the

- GEM5\_RESOURCE\_JSON environment variable to point to the JSON, if you want to just use the resources in the JSON.
- GEM5\_RESOURCE\_JSON\_APPEND environment variable to point to the JSON, if you want to use local resources along with gem5 resources.

For more details on how to use local resources, read [local resources documentation](#)



# Why use local resources

gem5 has two main ways to use local resources.

- Directly create the resource object by passing the local path of the resource
  - `BinaryResource(local_path=/path/to/binary)`
  - We can use this method when we are making new resources and want to quickly test the resource
- If we are going to use or share the resource that we created, it is better to create a JSON file and update the data source as mentioned in the above slide.
  - with this method we can use `obtain_resource`
  - This method makes the simulations more reproducible and consistent.

Lets do an example that create a local binary and runs that binary on gem5.



# Lets create a binary

First lets write a [C program that prints a simple triangle pattern](#).

```
gcc -o pattern.c
```

Lets compile this program. This will be the binary that we will run in gem5.

First lets use the local path method [gem5 config](#)

lets create the binary resource object

- `BinaryResource(local_path="path")`

lets run the simulation and see the output

# Lets create a JSON file for the binary resource

The JSON for the binary would look like this:

## JSON resource

```
{
  "category": "binary",
  "id": "x86-pattern-print",
  "description": "A simple X86 binary that prints a pattern",
  "architecture": "X86",
  "size": 1,
  "tags": [],
  "is zipped": false,
  "md5sum": "2a0689d8a0168b3d5613b01dac22b9ec",
  "source": "",
  "url": "file://./pattern",
  "code_examples": [],
  "license": "",
  "author": [
    "Harshil Patel"
  ],
  "source_url": "",
  "resource_version": "1.0.0",
  "gem5 versions": [
    "23.0",
    "23.1",
    "24.0"
  ],
  "example_usage": "obtain_resource(resource_id=\"x86-pattern-print\")"
}
```



# Lets get the resource and run the simulation

We get the binary by using obtain resource

```
board.set_se_binary_workload(obtain_resource("x86-pattern-print"))
```

lets run the simulation

To run the simulation we use the normal run command but add

```
GEM5_RESOURCE_JSON_APPEND=../../materials/02-Using-gem5/02-gem5-resources/complete/03-local-resources/run-local-resource-json.py
```

before the build/X86/gem5.opt

