# BFTGym: An Interactive Playground for BFT Protocols

Haoyun Qin
University of Pennsylvania
qhy@seas.upenn.edu

Chenyuan Wu
University of Pennsylvania
wucy@seas.upenn.edu

Mohammad Javad Amiri
Stony Brook University
amiri@cs.stonybrook.edu

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

## ABSTRACT

Byzantine Fault Tolerant (BFT) protocols serve as a fundamental yet intricate component of distributed data management systems in untrustworthy environments. BFT protocols exhibit different design principles and performance characteristics under varying workloads and fault scenarios. The proliferation of BFT protocols and their growing complexity have made it increasingly challenging to analyze the performance and possible application scenarios of each protocol. This demonstration showcases *BFTGym*, an interactive platform that allows audience members to (1) evaluate, compare, and gather insights into the performance of various BFT protocols under a wide range of conditions, and (2) prototype new BFT protocols rapidly.

**PVLDB Reference Format:**
Haoyun Qin, Chenyuan Wu, Mohammad Javad Amiri, Ryan Marcus, and Boon Thau Loo. BFTGym: An Interactive Playground for BFT Protocols. PVLDB, 17(12): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/JeffersonQin/BFTGym.

## 1 INTRODUCTION

Byzantine Fault Tolerant (BFT) protocols are crucial in distributed data management systems running in untrustworthy environments. At their core, these systems utilize State Machine Replication (SMR), ensuring that non-faulty replicas execute client requests in the same order despite the concurrent failure of at most $f$ Byzantine replicas.

The existing landscape of BFT protocols demonstrates remarkable complexity. These protocols [1, 3–7] vary in numerous aspects, including their authentication techniques, phase structures, communication patterns such as multicast or linear gathering, and view change processes. This diversity in protocol design yields distinct performance under varying workloads and fault scenarios. Using Zyzzyva [6] as an example, when faced with non-responsive nodes, it shifts from an optimistic and linear fast path to an expensive slow path with one more phase. This performance degradation is in sharp contrast to other protocols like PBFT [3], which not only would not degrade, but may experience better throughput due to fewer messages and resource consumption.

To address the complexities of the BFT protocol landscape, we have developed Bedrock [2], a comprehensive framework for BFT protocol analysis and implementation. Bedrock stands out in its ability to facilitate fair comparisons across a broad spectrum of BFT protocols by minimizing disparities stemming from different implementation factors, such as programming languages, software libraries, and runtime environments. The framework encompasses implementations of a wide range of existing BFT protocols, including notable examples like [1, 3–7]. To enable rapid prototyping of new protocols, Bedrock allows users to define protocol logic in a Domain Specific Language (DSL). For unique requirements that can not be expressed by the built-in DSL, Bedrock enables users to integrate custom plugins into the framework.

Building upon Bedrock's solid foundation, we introduce *BFTGym*, a platform designed to further simplify and enhance the *experimentation* of BFT protocols. While Bedrock provides the analytical and implementation backbone, BFTGym focuses on minimizing the experimental burden and maximizing interactivity and user engagement. BFTGym features an intuitive and *interactive* user interface, which accepts users' specifications for client workloads, system settings and fault scenarios. At the same time, the interface provides users with instant feedback and real-time performance visualizations. We believe such an interactive tool and seamless experimentation experience are beneficial to the database systems community, seeing how out-of-the-box experimentation tools and platforms such as Hugging Face, Jupyter, and Gradio has fostered the development in the field of Machine Learning.

We will demonstrate BFTGym and its capabilities through two demonstration scenarios. The first one explores the functionality of BFTGym as an interactive platform. Here, system developers who want to choose a suitable consensus protocol for their system are empowered to dynamically adjust workloads and inject faults, with the platform providing immediate performance feedback. This interactive aspect is further enhanced by a comparison panel, which facilitates the analysis and comparison of different experimental trials, helping to understand the complex interactions between various workloads, fault scenarios, and the behaviors of different BFT protocols. The second one focuses on how BFTGym aids the development of new BFT protocols. By leveraging Bedrock's easy-to-use DSL and its flexible system architecture, BFTGym offers consensus experts a conducive environment for the rapid prototyping of new BFT protocols. This aspect of BFTGym not only streamlines the development process, but also provides a testing ground for the practical viability and performance of new protocols.


Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:XX.XX/XXX.XX

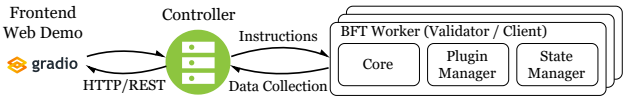



**Figure 1: Overview of BFTGym.**

Through the demonstration of the two cases, we hope to expose our audience to an accessible and engaging platform to navigate the intricate landscape of BFT protocols, fostering a more hands-on approach to the exploration and development of BFT systems.

## 2 SYSTEM OVERVIEW

In this section, we give an overview of BFTGym. As shown in Figure 1, the architecture of BFTGym consists of three interconnected components: the Front-end demo page, the Controller, and the Bedrock BFT Workers. Each of these components plays a critical role in the seamless operation and user interaction with the system.

**Front-end Web Demo Page.** The front-end builds upon Gradio and acts as the primary interface for users, offering an intuitive gateway to BFTGym. It is meticulously designed to facilitate user interaction, allowing for the adjustment of workload and fault configurations at run-time, the initiation of experiments, and the analysis and comparison of results. The detailed functionalities and user interaction will be further elaborated in Section 3.1.

**Controller.** The Controller serves as the relay between the front end and BFT workers. It orchestrates instance deployment, conveys configuration changes, and transmits instructions from the front end to the BFT workers, while also aggregating their runtime data. This integral connectivity is achieved through HTTP/RESTful APIs, ensuring a smooth and responsive communication flow.

**Bedrock BFT Workers.** Behind the scene of BFTGym are the BFT Workers of Bedrock [2]. These workers are deployed on a configurable cluster of distributed servers, facilitating studying how hardware setup affects protocol performance. Each worker plays the role of either a validator or a client, which comprises three essential components: the Plugin Manager, the State Manager, and the Core. The Plugin Manager is a versatile component, enabling the system to operate in a modular plug-and-play fashion. Its extensibility allows users to integrate custom plugins into the framework, when they have new protocol design choices that can not be captured by the built-in DSL. The State Manager tracks the states and transitions of each worker as defined in the Bedrock DSL for the specific protocol. Its responsibilities include tracking messages and calculating quorum conditions, ensuring the smooth phase execution and transitions of the protocol. The Core forms the kernel of the worker's functionality, encompassing the communication channel between different validators and clients. It also diligently tracks the execution of requests through various state variables such as views and sequence numbers, and handles intricate details like timers and watermarks. In summary, Bedrock BFT Workers underpins the robust and flexible nature of BFTGym.

## 3 DEMONSTRACTION SCENARIOS

In this section, we will introduce two use cases of BFTGym: one for interactive performance experimentation under different workloads and fault scenarios (Section 3.1) and the other for fast prototyping of new BFT protocols (Section 3.2).

### 3.1 Interactive Performance Experimentation

Figure 2 provides an overview of BFTGym's user interface. The tabs **A** and **B** on the top allow users to switch between the interactive playground view and the result comparison view.

**System Configurations C.** The BFTGym platform enables users to configure various fault scenarios and workload conditions. **C.1** caters to the specification of faults with two options: (1) non-responsive faults, where nodes fail to participate in the consensus process, and (2) slowness attacks, which are particularly detrimental in leader-based BFT protocols. In such protocols, the delay of a leader's proposals might seriously affect system performance. A compromised leader may manipulate proposal timings to degrade system throughput and latency subtly enough to avoid triggering a view change. These slowdowns may also stem from an overburdened leader or a leader with poor hardware resources, albeit with less severity. **C.2** adjusts the typical key-value store workload parameters, including contention level, dataset size, and sizes of request and reply messages, alongside emulated computation complexity (i.e., additional cycle consumption), workload mixture, and read-only request proportion. Utilizing Cloudlab as the infrastructure, BFTGym enables painless deployment setup via **C.3** by specifying Cloudlab credentials, experiment and cluster profile name, which also allows users to instantiate and compare experiments among different hardware setups in a single unified interface.

**Control Panel D.** The Control Panel serves as the operational hub for BFTGym, where users label each trial with a name tag and select the protocol to be executed in this trial. BFTGym supports six built-in protocols: PBFT [3], CheapBFT [5], Zyzzyva [6], Prime [1], HotStuff-2 [7], and SBFT [4]. Notably, this panel also accommodates custom protocols implemented by users, which will be elaborated in Section 3.2. The trial is initiated and terminated using the start and stop buttons at the bottom.

**Results Visualization G.** BFTGym provides real-time performance visualization during active trials. Upon trial initiation, **G.1** illustrates system throughput with respect to time, while **G.2** displays the latest committed sequence number across all replicas. This feature is particularly insightful as it can disclose the presence of non-responsive faults through the visual discrepancies in the sequence numbers. To visualize past trials, users can click on dropbox to select a trial and use the button on the right to refresh.

**Dynamic Configurations E and Status Indicators F.** Users have the flexibility to change the configurations of **C.1** and **C.2** dynamically during trials. Changes are applied to the current trial upon clicking the update button **E**. Each BFT worker periodically polls the controller for the latest configuration file, with **F** signaling the successful application of new configurations for each worker.

**Results Comparison View H, I.** The comparison view of BFTGym, encapsulated within tabs **H** for session selection and **I** for graphical output, is critical for the empirical analysis of BFT protocols. Users can select various sessions to compare using **H** on the upper side. The lower side of the view, annotated by **I**, depicts the throughput of different trials in one graph, facilitating a direct

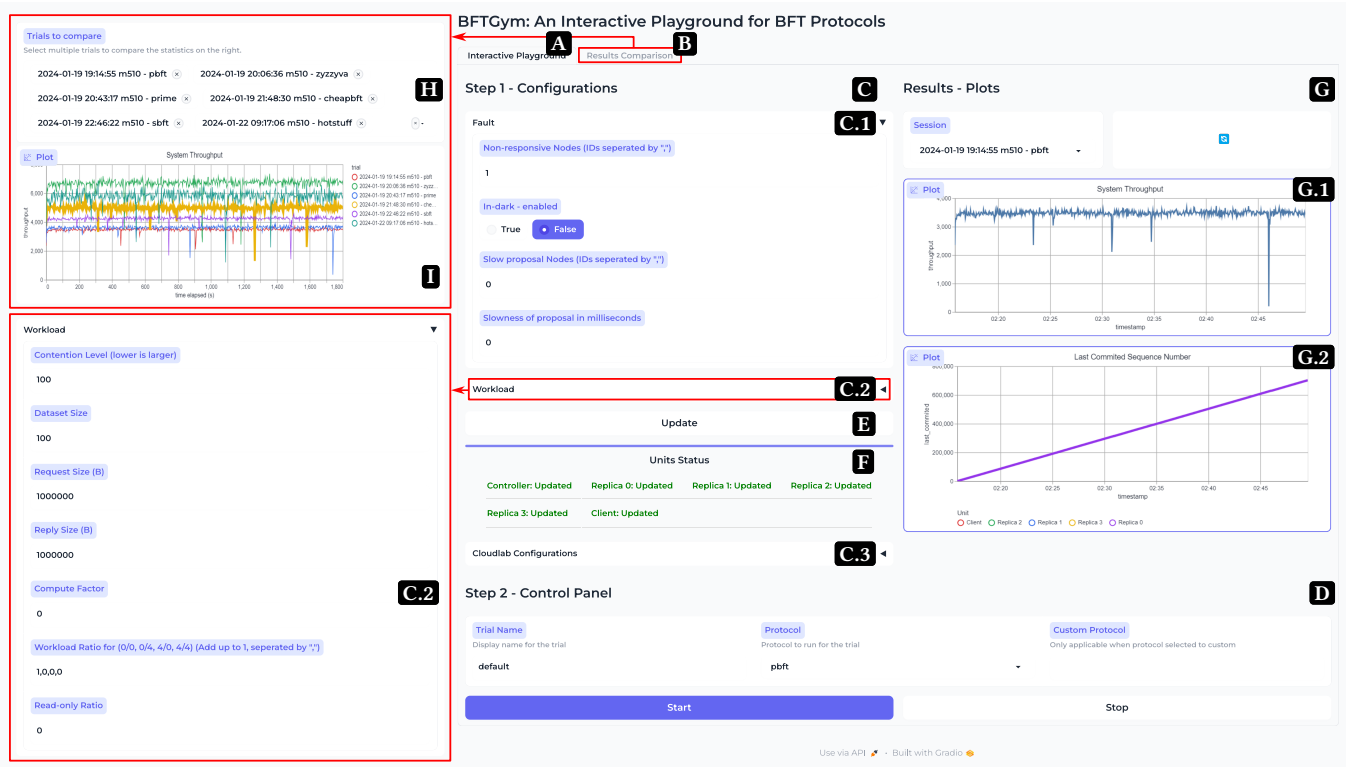
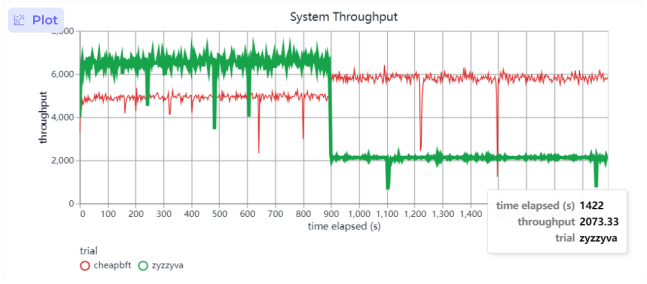



**Figure 2: Screenshot of BFTGym's user interface.**

performance comparison across protocols. The plot **I** in Figure 2 illustrates a comparative analysis for six built-in protocols conducted on the Cloudlab `m510` cluster under $f = 1$ and a standard 0/0 workload (0KB request and reply size), where we have this typical performance ranking in descending order of Zyzzyva, Hotstuff-2, CheapBFT, SBFT, Prime and PBFT under this fault-free scenario.

Below, we further highlight two insightful performance comparisons we found using BFTGym. As the first example, Figure 3(a) shows how non-responsive faults affect protocols' performance. The experiment is performed under a standard 0/0 workload on a 4-machine Cloudlab `m510` cluster for 30 minutes, with a non-responsive fault started in the middle. We discover that Zyzzyva's [6] throughput dropped significantly due to its speculative execution design, while CheapBFT [5] received a performance gain given less number of messages to process. The second example compares the performance robustness of protocols when the slowness fault takes place. The system is configured with a 1/0 standard workload with 20-millisecond leader slowness. As shown in Figure 3(b), to our surprise, the winner is not Prime [1], which is specifically designed to handle slowness attacks by proactive leader replacement. Rather, HotStuff-2 [7] renders the best performance, benefiting from its linear responsive view-change and leader rotation, amortizing the cost of one slow leader to all nodes across the cluster.



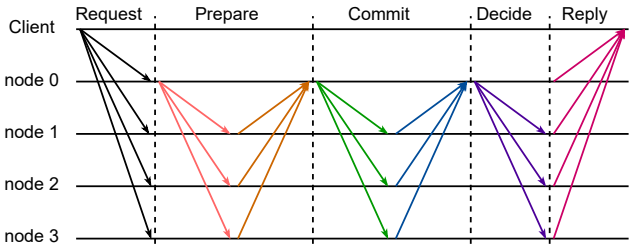

(a) 0/0 standard workload with non-responsive fault launched at the middle


(b) 1/0 standard workload with 20 millisecond slowness fault

**Figure 3: Two insightful performance comparisons found using BFTGym.**

## 3.2 Fast Protocol Prototyping

The unique strength of BFTGym is to expedite the prototyping process for new BFT protocols, by leveraging the versatility of




```
# protocol properties
general:
  leader: rotate
  request-target: primary

# roles involved
roles:
  - primary
  - nodes
  - client

# phase definitions
phases:
  - name: normal
    states:
      - idle
      - wait_prepare
      - wait_commit_primary
      - wait_commit_all
      - wait_decide
      - executed
    messages:
      - name: request
        request-block: true
      - name: reply
        request-block: true
      - name: prepare
        request-block: true
      - commit
      - decide

  - name: checkpoint
    messages:
      - checkpoint

# data flow
transitions:
  from:
    - role: client
      state: idle
      to:
        - state: executed
          update: sequence
          condition:
            type: message
            message: reply
            quorum: f + 1

    - role: primary
      state: idle
      to:
        - state: wait_prepare
          condition:
            type: message
            message: request
            quorum: 1
          response:
            - target: nodes
              message: prepare

- role: nodes
  state: idle
  to:
    - state: wait_commit_all
      condition:
        type: message
        message: prepare
        quorum: 1
      response:
        - target: primary
          message: prepare
- role: primary
  state: wait_prepare
  to:
    - state: wait_commit_primary
      condition:
        type: message
        message: prepare
        quorum: 2f + 1
      response:
        - target: nodes
          message: commit
- role: nodes
  state: wait_commit_all
  to:
    - state: wait_decide
      condition:
        type: message
        message: commit
        quorum: 1
      response:
        - target: primary
          message: commit
- role: primary
  state: wait_commit_primary
  to:
    - state: executed
      update: sequence
      condition:
        type: message
        message: commit
        quorum: 2f + 1
      response:
        - target: nodes
          message: decide
        - target: client
          message: reply
- role: nodes
  state: wait_decide
  to:
    - state: executed
      update: sequence
      condition:
        type: message
        message: decide
        quorum: 1
      response:
        - target: client
          message: reply
```

**Figure 4: HotStuff-2 communication pattern and its Bedrock DSL. Corresponding code snippet and message patterns are annotated using the same color. Fully annotated version available here: https://haoyunqin.com/files/hotstuff-2-dsl.yaml.**

the Bedrock [2] DSL. We will exemplify this functionality through Hotstuff-2 [7], a recently proposed BFT protocol, demonstrating the streamlined workflow for protocol development.

Bedrock DSL enables protocol developers to express the specifications of a BFT protocol succinctly. As shown in Figure 4, the protocol is defined in terms of general properties, roles, phases, and transitions. By HotStuff-2's design, the protocol is configured to rotate its leader every a few sequence numbers. Roles involved here are the `primary`, `nodes` (i.e. replicas), and the `client`. The phase configuration is expressed through states e.g. `idle`, `wait_prepare`, and `wait_commit_primary`, along with messages e.g. `request`, `reply`, `prepare`, and `commit`. The logical flow of the protocol is further detailed in the transition section, delineating the conditions for state changes. For instance, a node transits from `idle` to `wait_commit_all` upon receiving a `prepare` message, and will subsequently issue a `prepare` message to the `primary`. All transitions in the code are highlighted using the same color as in the protocol diagram.

The protocol prototype is then made available in BFTGym within three steps: (1) place the protocol DSL code under the configuration directory; (2) select `custom` through the protocol dropdown box in **D**; (3) enter the protocol name in the custom protocol field. Once selected, the custom protocol is seamlessly integrated into the experimental workflow, benefiting from the same functionalities as other built-in protocols, such as initiating and stopping trials, and engaging in comparative analysis with existing protocols.

## 4 CONCLUSION

In this paper, we presented BFTGym, an interactive playground for BFT protocols. Our case studies have demonstrated that BFTGym can effectively serve as an interactive platform tailored for system developers, and a fast prototyping tool for consensus experts. We believe BFTGym will enhance understanding and innovation in BFT consensus within the distributed data management community.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback and suggestions. This work is funded by NSF grants CNS-2104882.

## REFERENCES

[1] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine replication under attack. *Transactions on Dependable and Secure Computing* 8, 4 (2011), 564–577.

[2] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2024. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 371–400.

[3] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, Vol. 99. USENIX Association, 173–186.

[4] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE/IFIP, 568–580.

[5] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: resource-efficient byzantine fault tolerance. In *European Conf. on Computer Systems (EuroSys)*. ACM, 295–308.

[6] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. *Operating Systems Review (OSR)* 41, 6 (2007), 45–58.

[7] Dahlia Malkhi and Kartik Nayak. 2023. HotStuff-2: Optimal Two-Phase Responsive BFT. *Cryptology ePrint Archive* (2023).