Event-Driven Programming II

Ryan Eberhardt and Julio Ballista

June 1, 2021

Logistics

- You're so close to the finish line!! 🏁 🏁
  - We're so proud of everything you've learned this quarter, and we hope you are as well!
- Please fill out the official course survey whenever you get a chance
  - This class has no departmental support and no certain future
  - If you felt that the material you learned this quarter was important for your growth, please indicate this on the survey!
  - If you felt we could have done a better job, please help us improve!

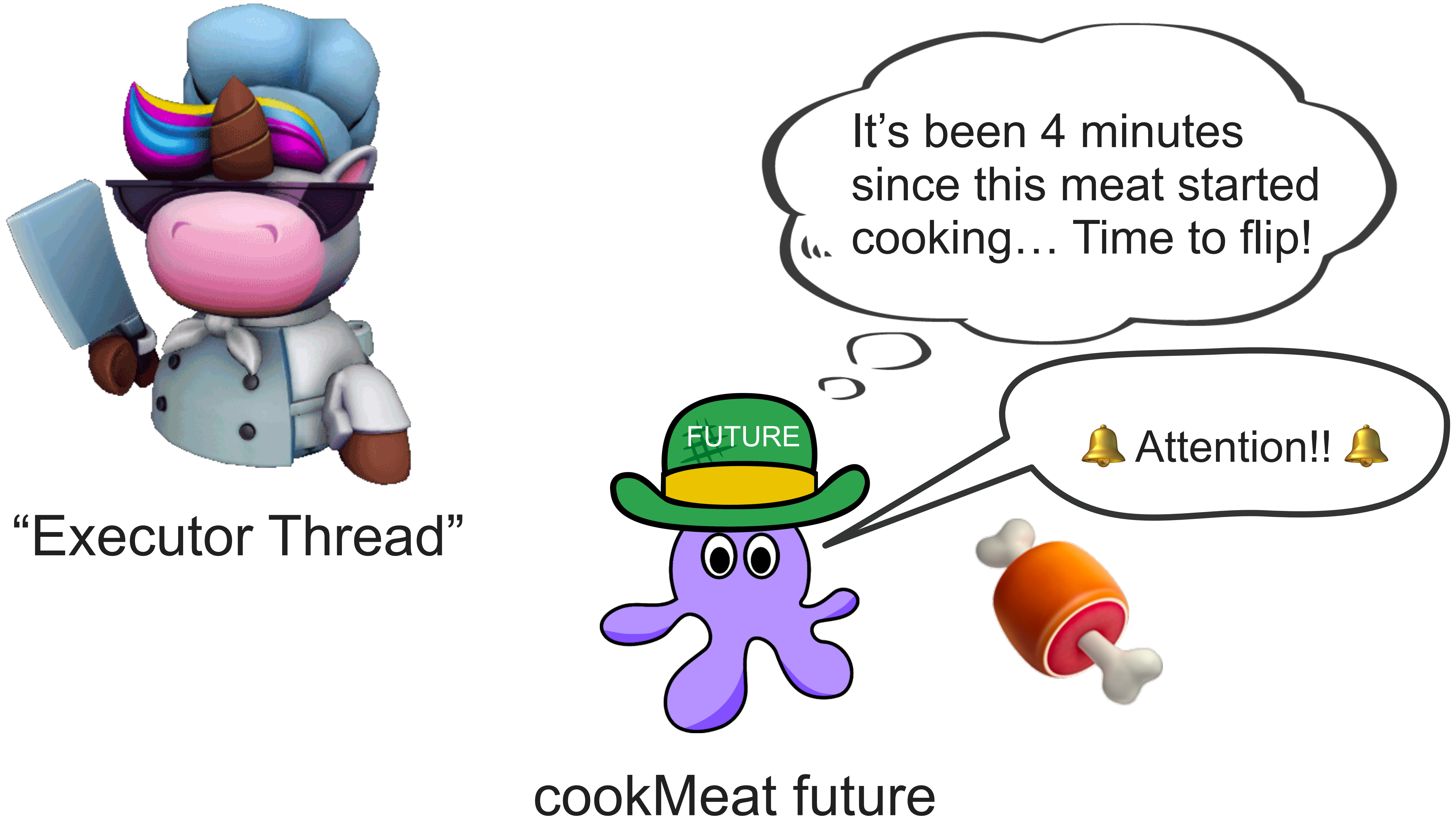
Roadmap

- Threads are great!
- But we can’t have too many of them, and context switches are expensive
- Event driven programming is nice in theory, but managing state seems hard
- Futures help us encapsulate state for each in-progress operation, making event-driven programming cleaner and more practical!
- Today: new syntax for making programming with futures even easier

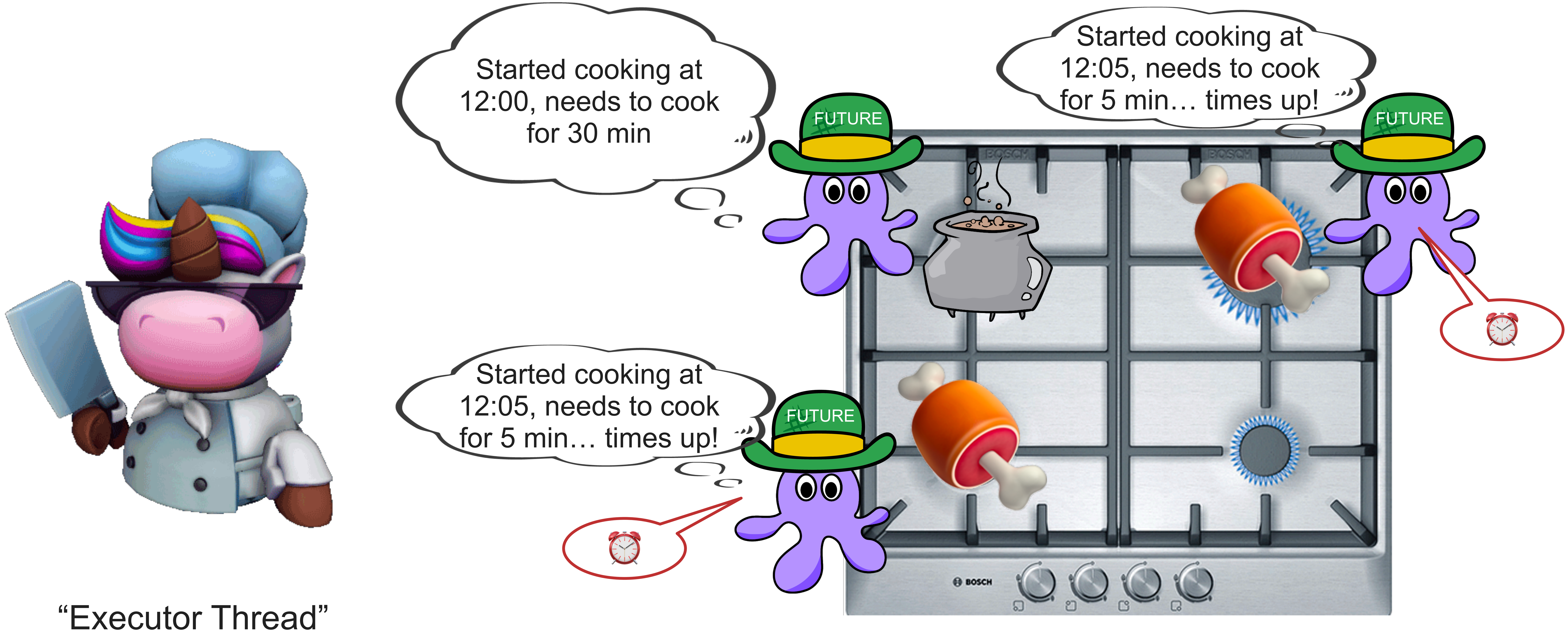
What are futures?

What are futures?

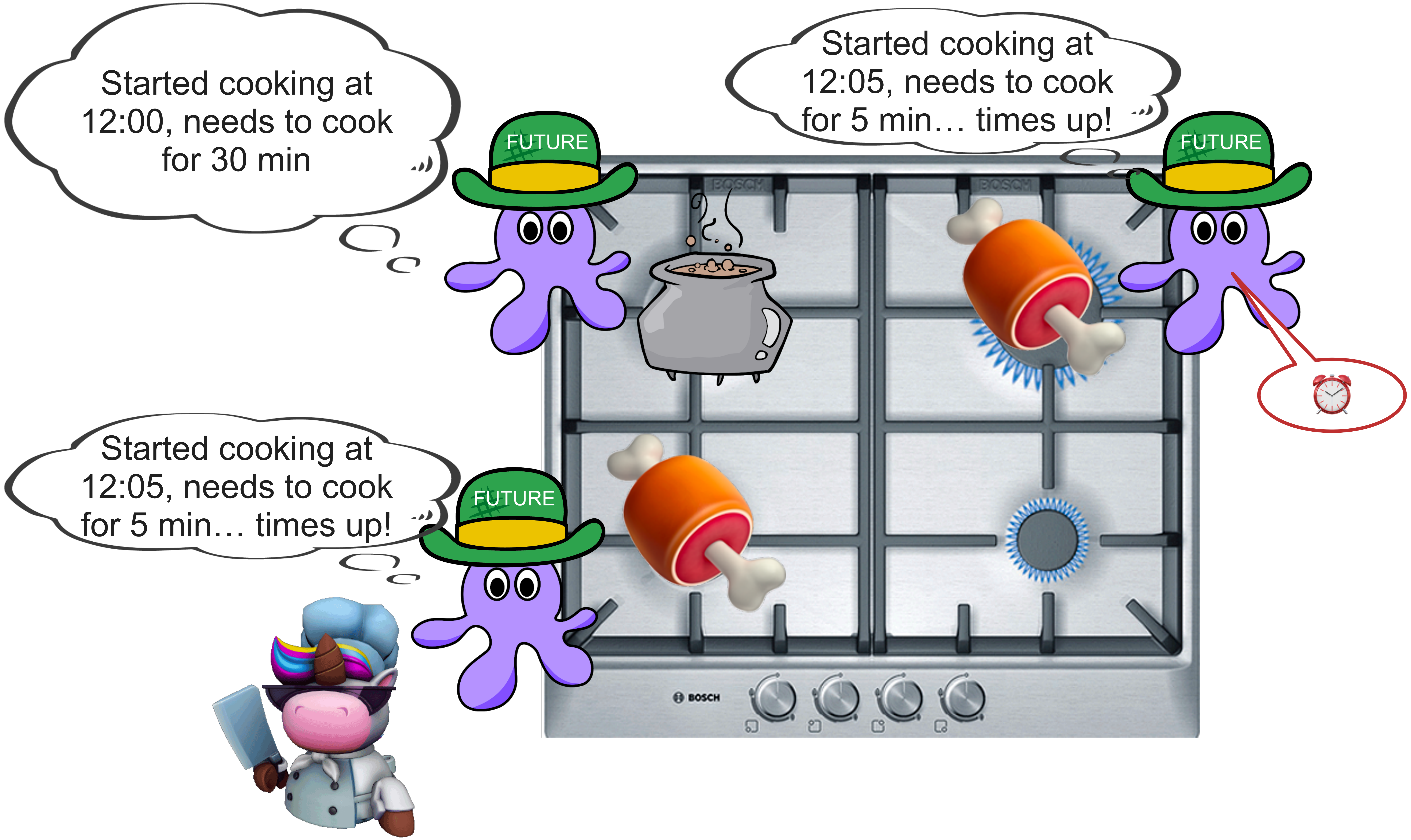
- Rust docs: “Futures are single eventual values produced by asynchronous computations.”
- You can think of a future as a helper friend that oversees each operation, remembering any associated state



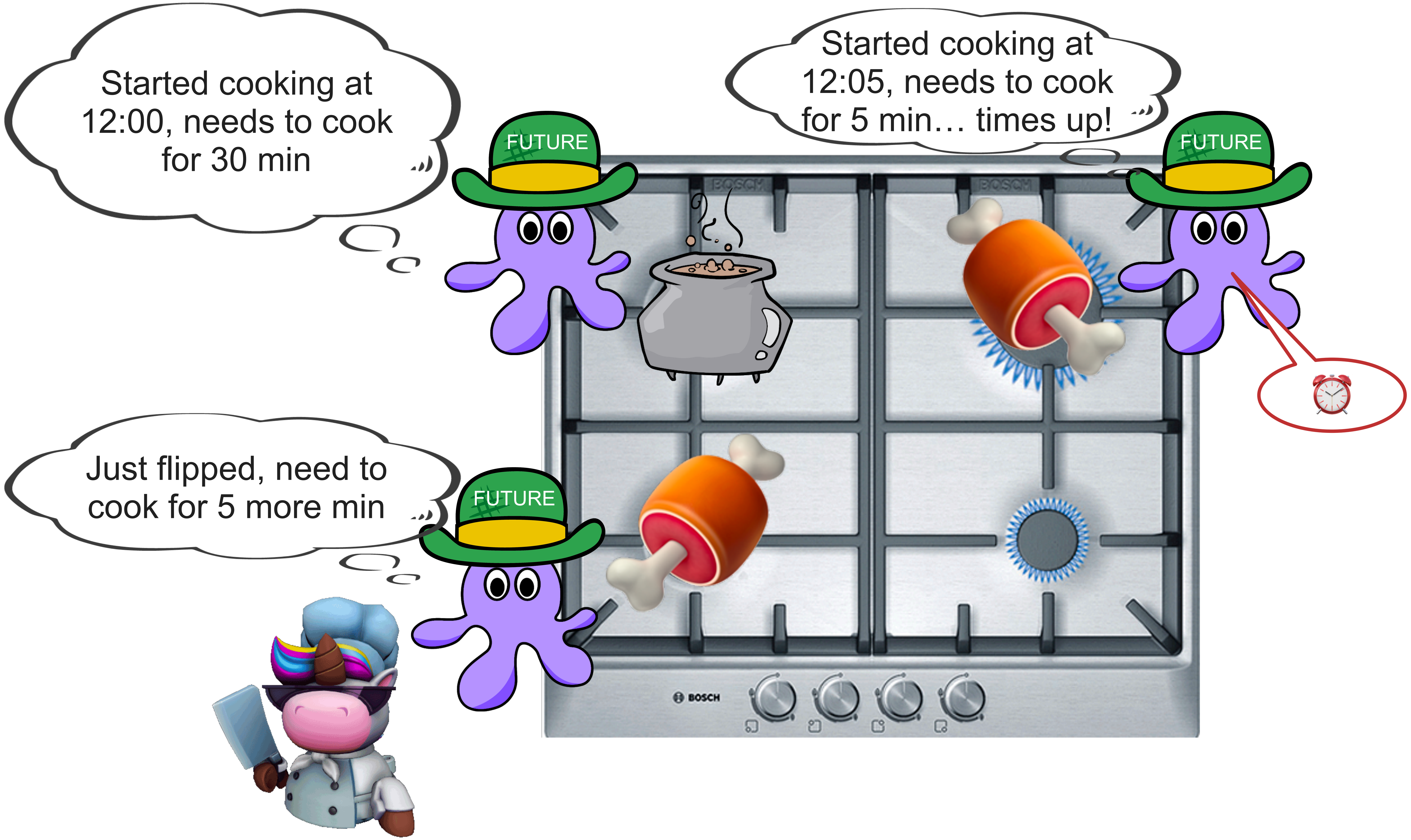
Futures Visualized



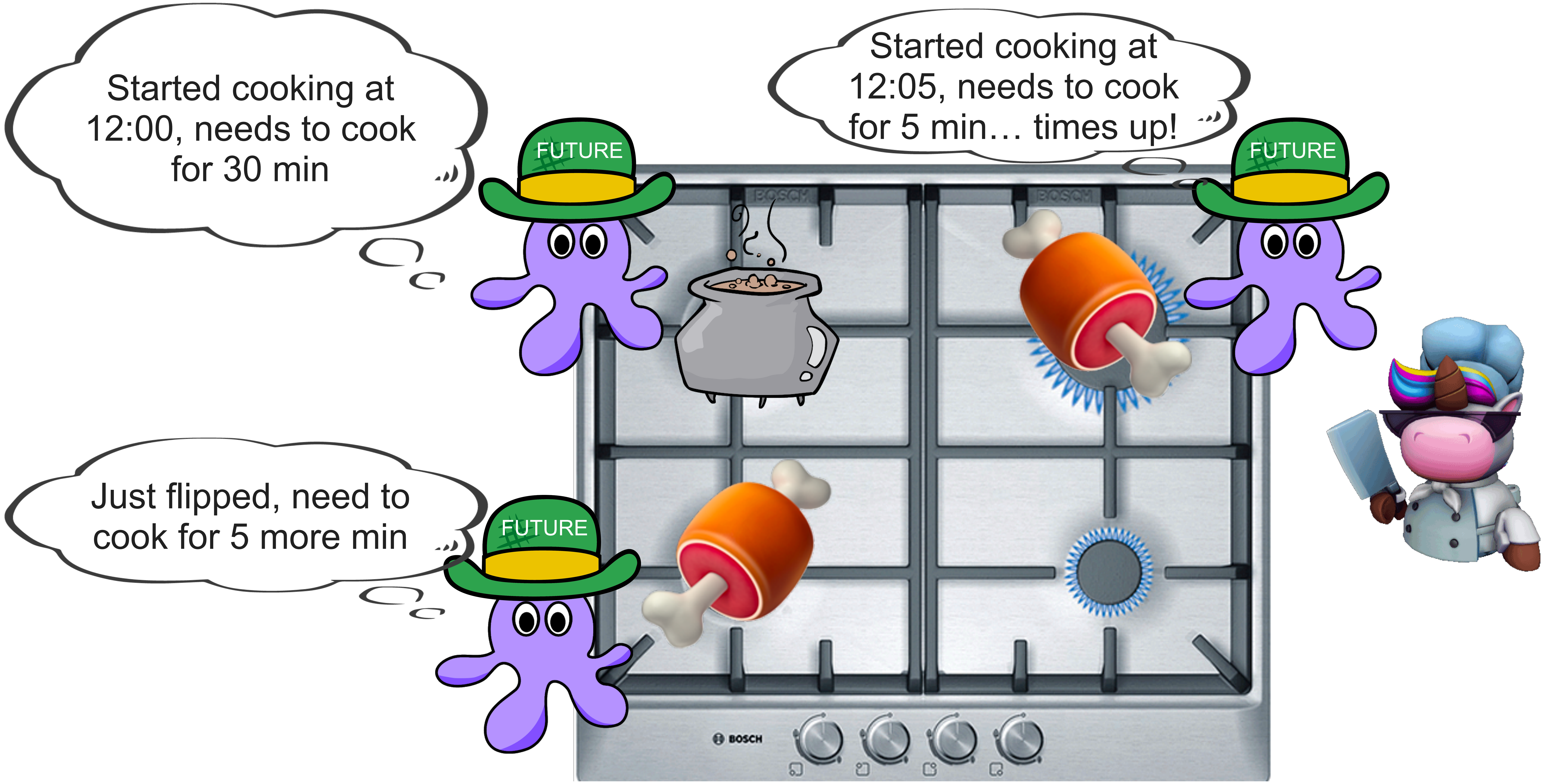
Futures Visualized



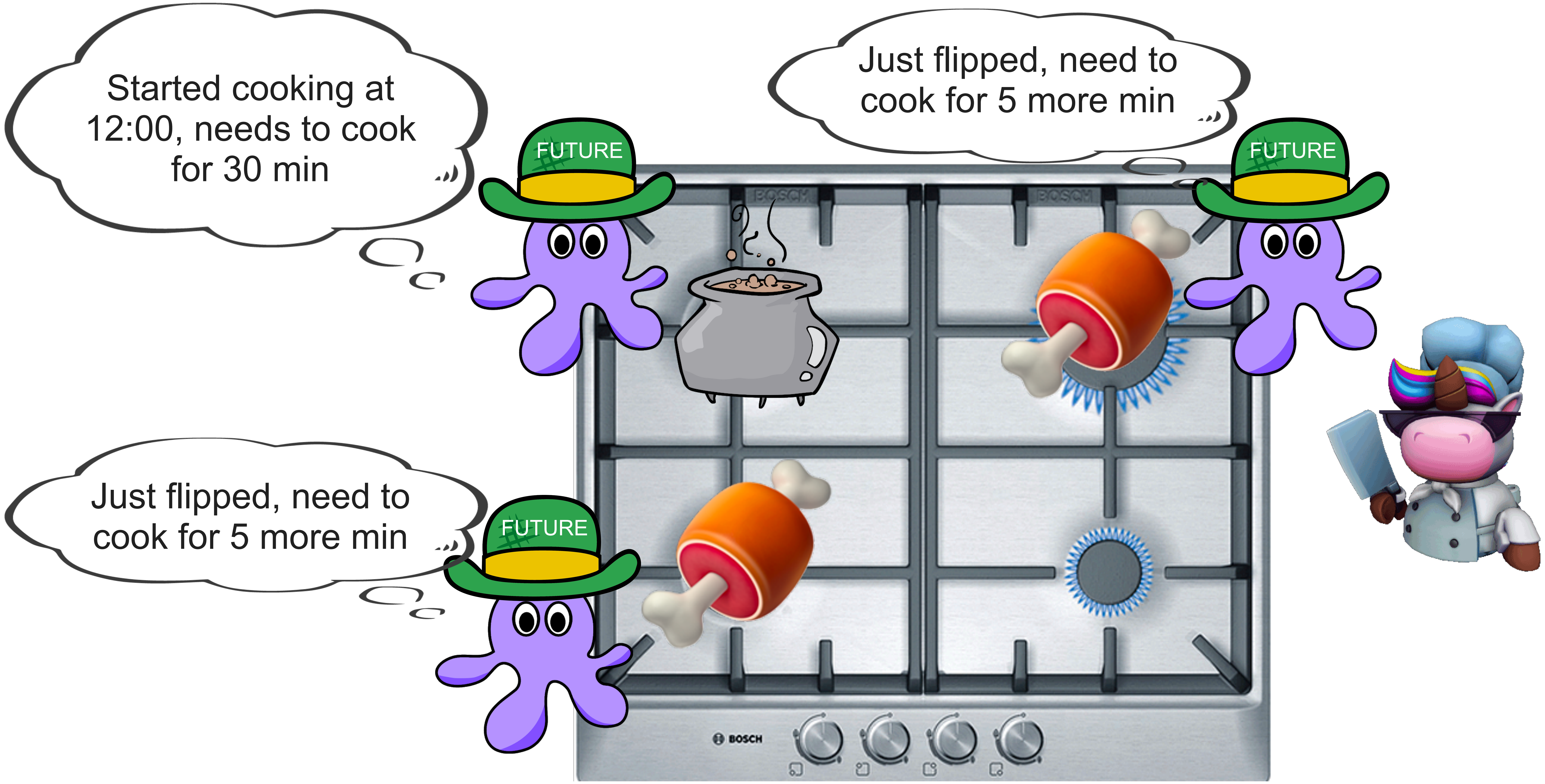
Futures Visualized



Futures Visualized

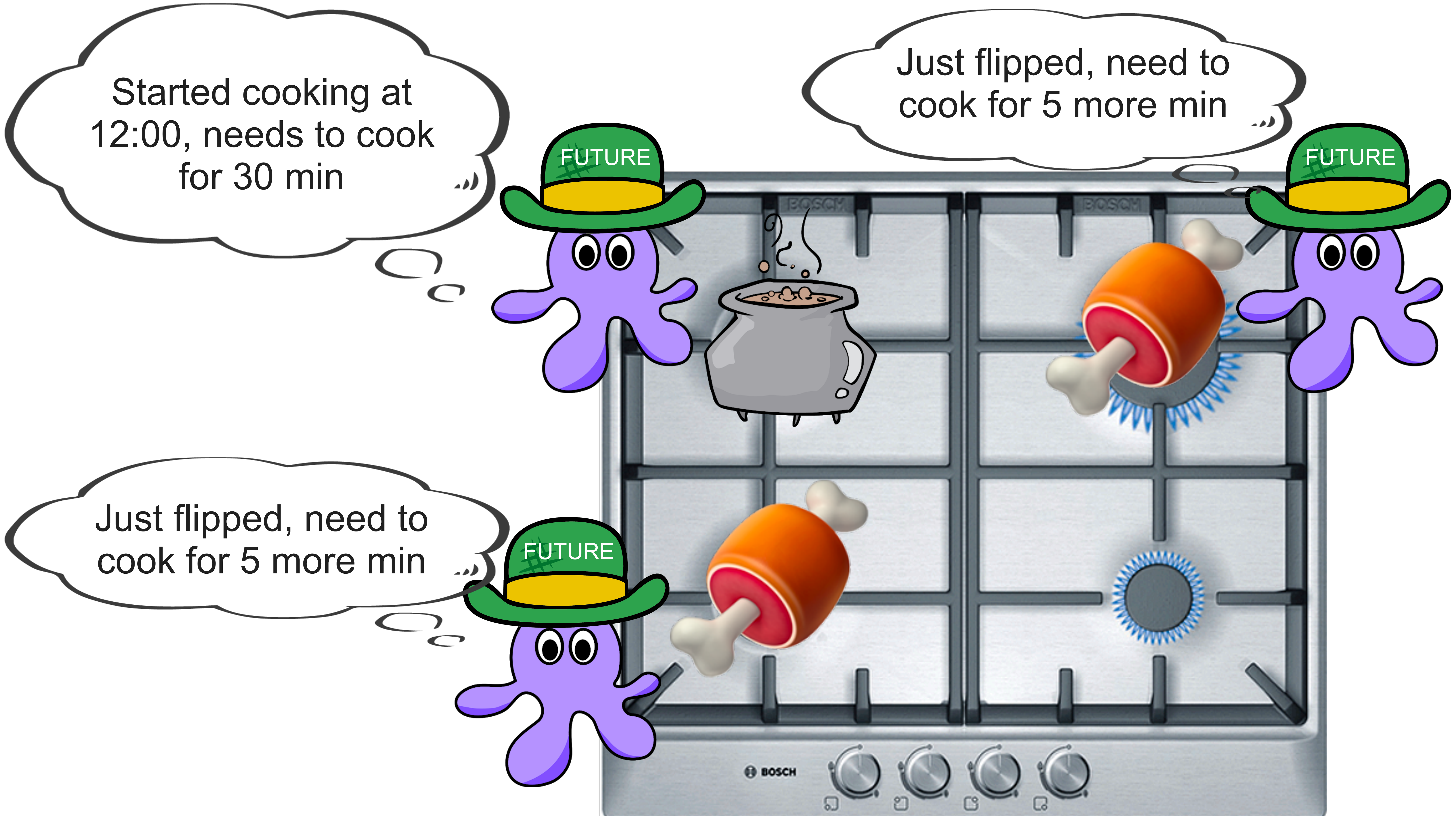


Futures Visualized



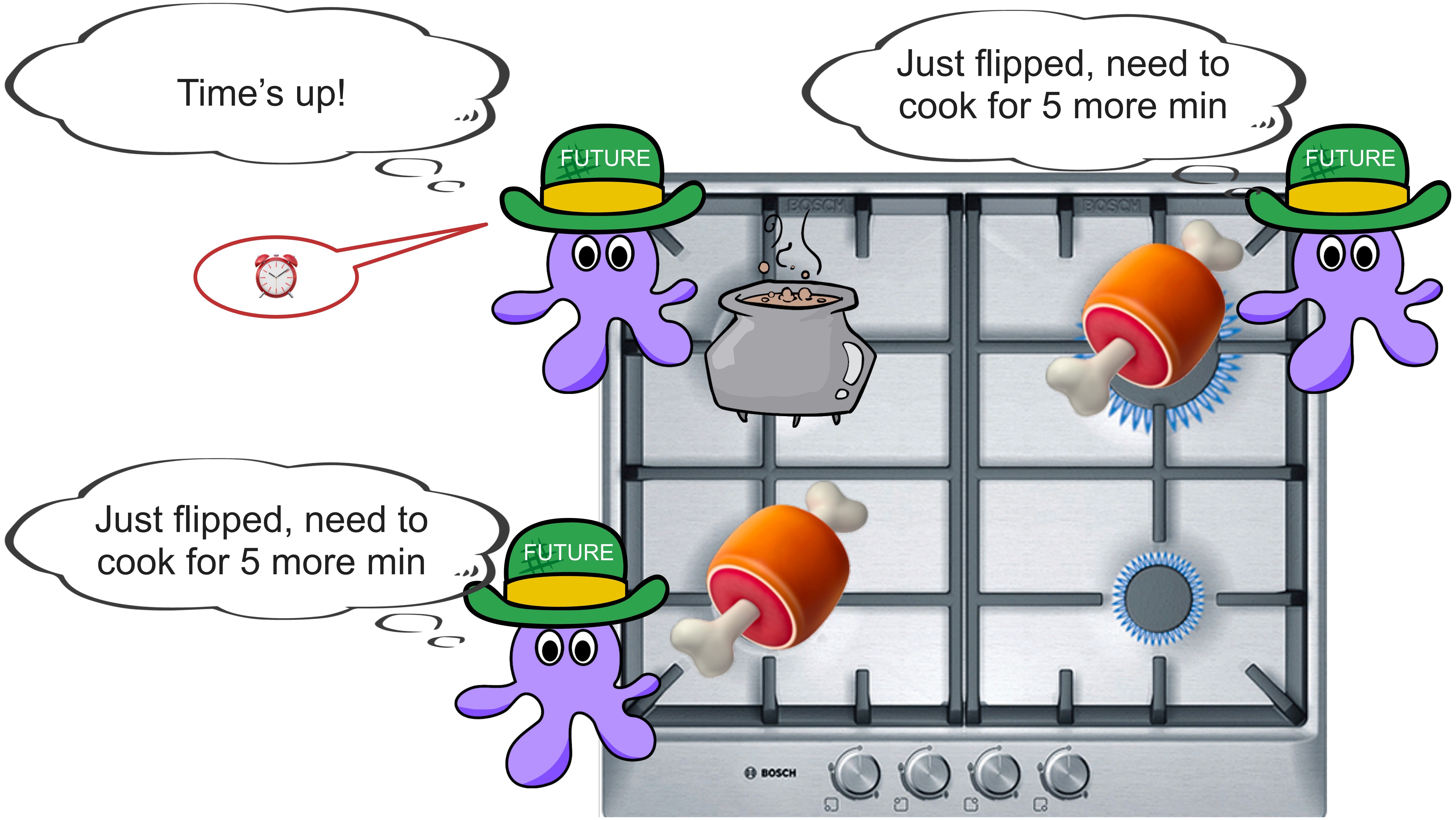
Futures Visualized

Executor thread
(sleeping)

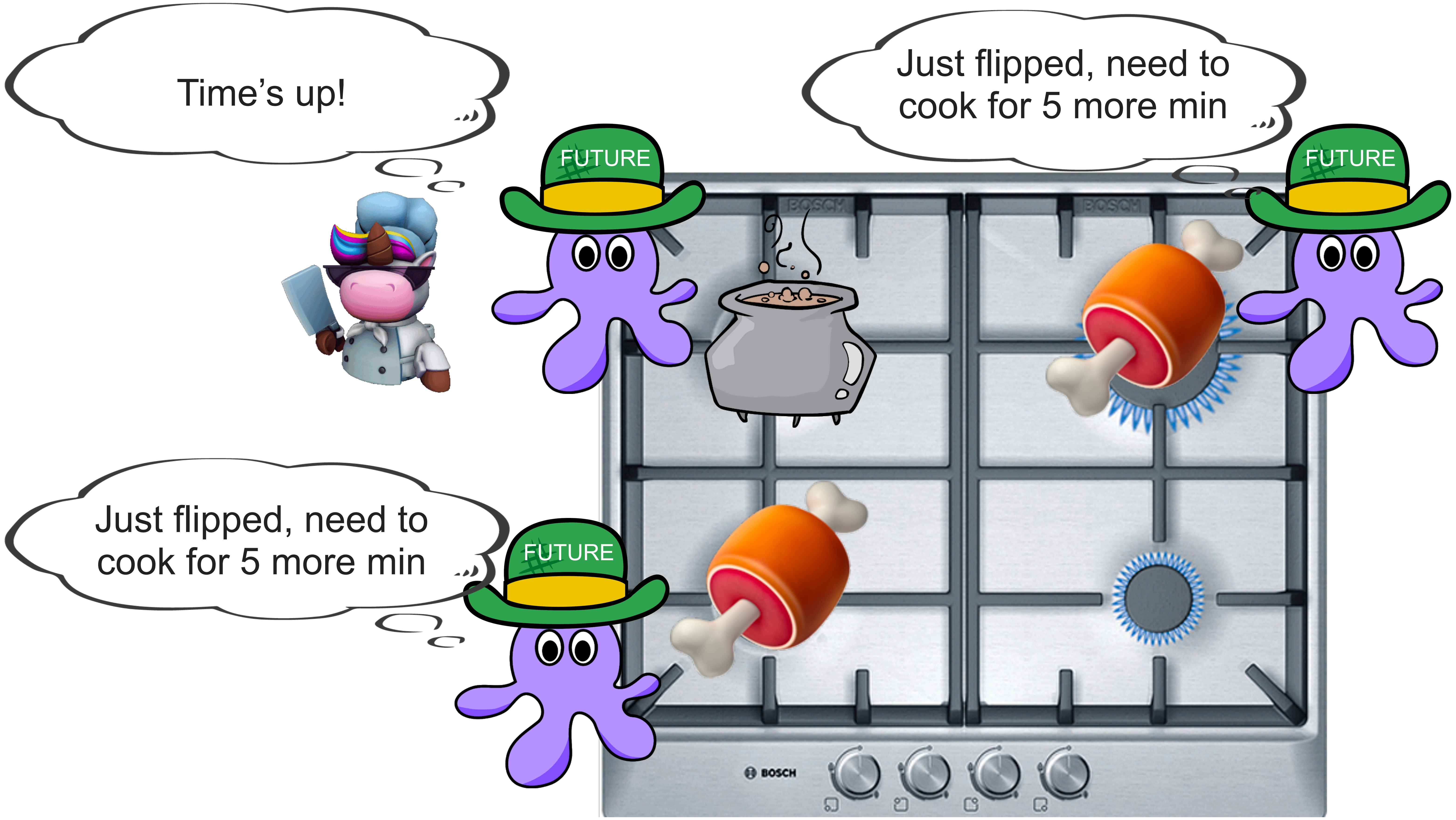


Futures Visualized

Executor thread
(sleeping)

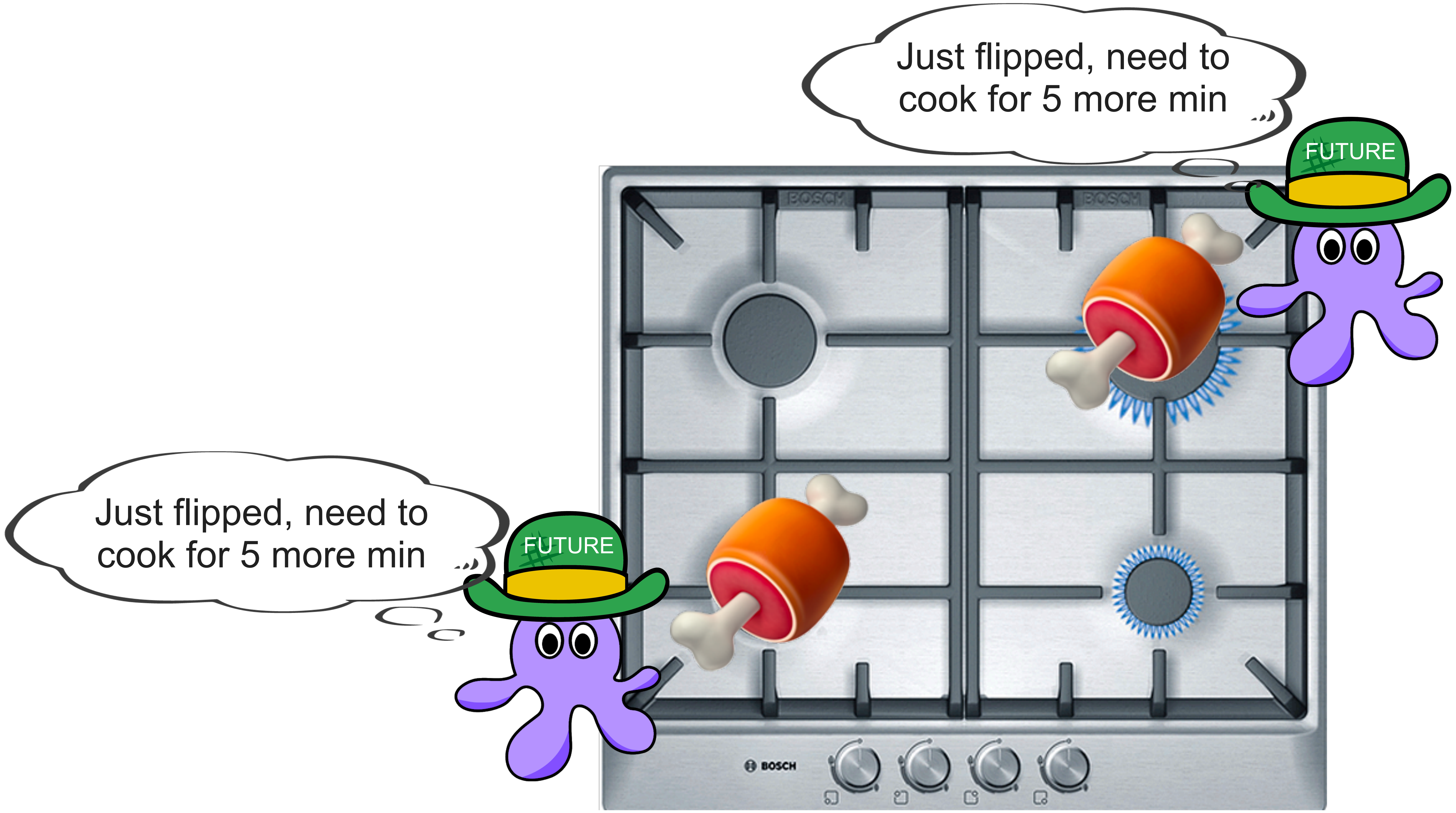


Futures Visualized

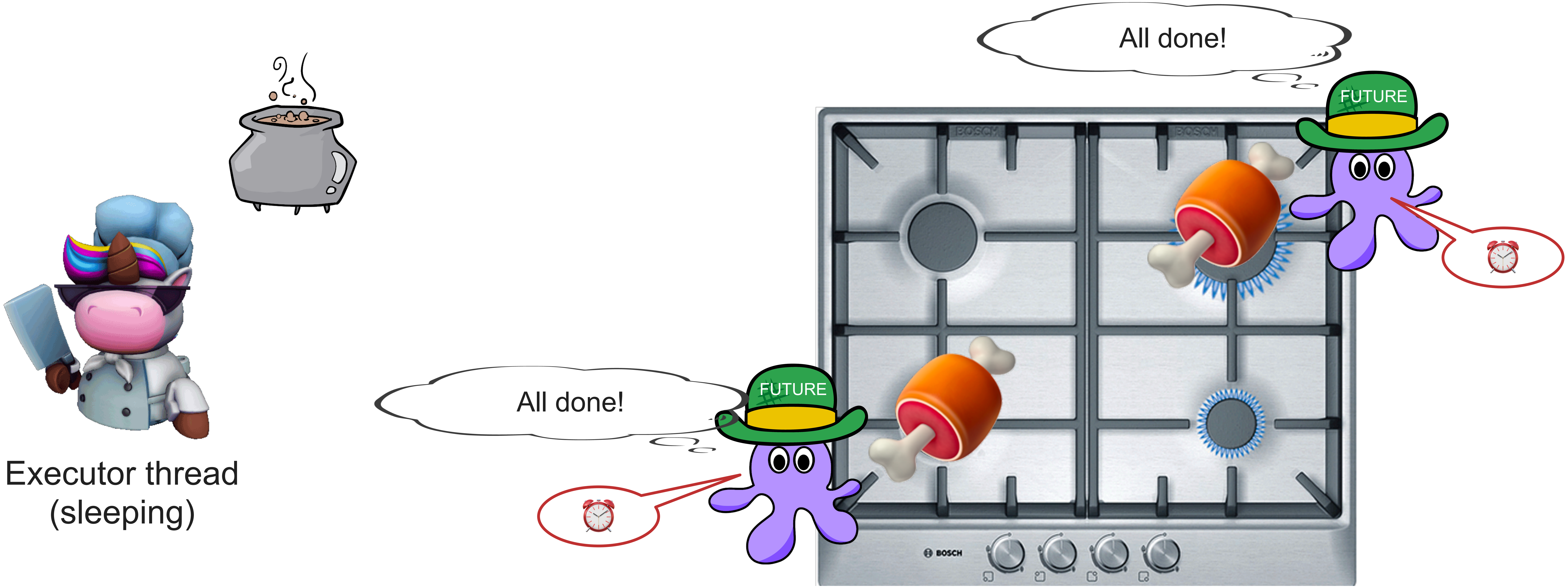


Futures Visualized

Executor thread
(sleeping)

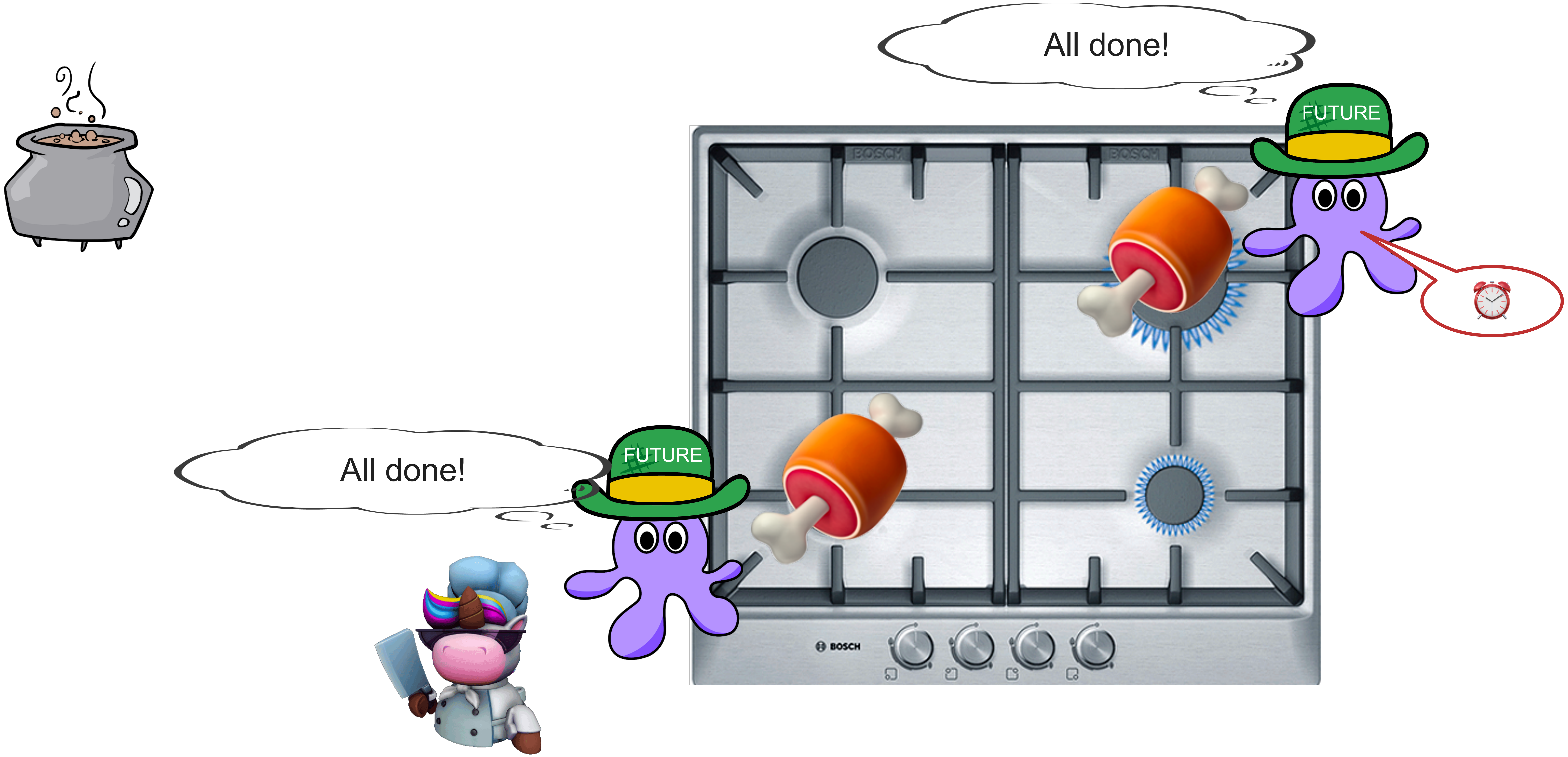


Futures Visualized



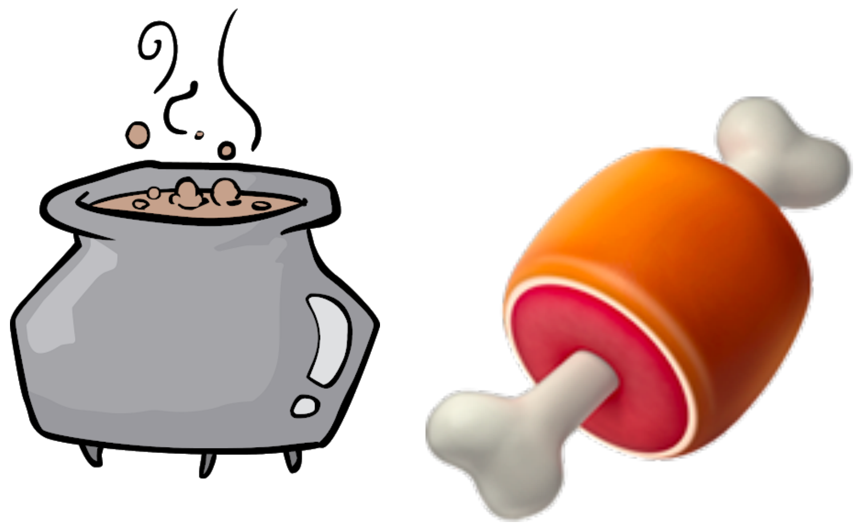
Executor thread
(sleeping)

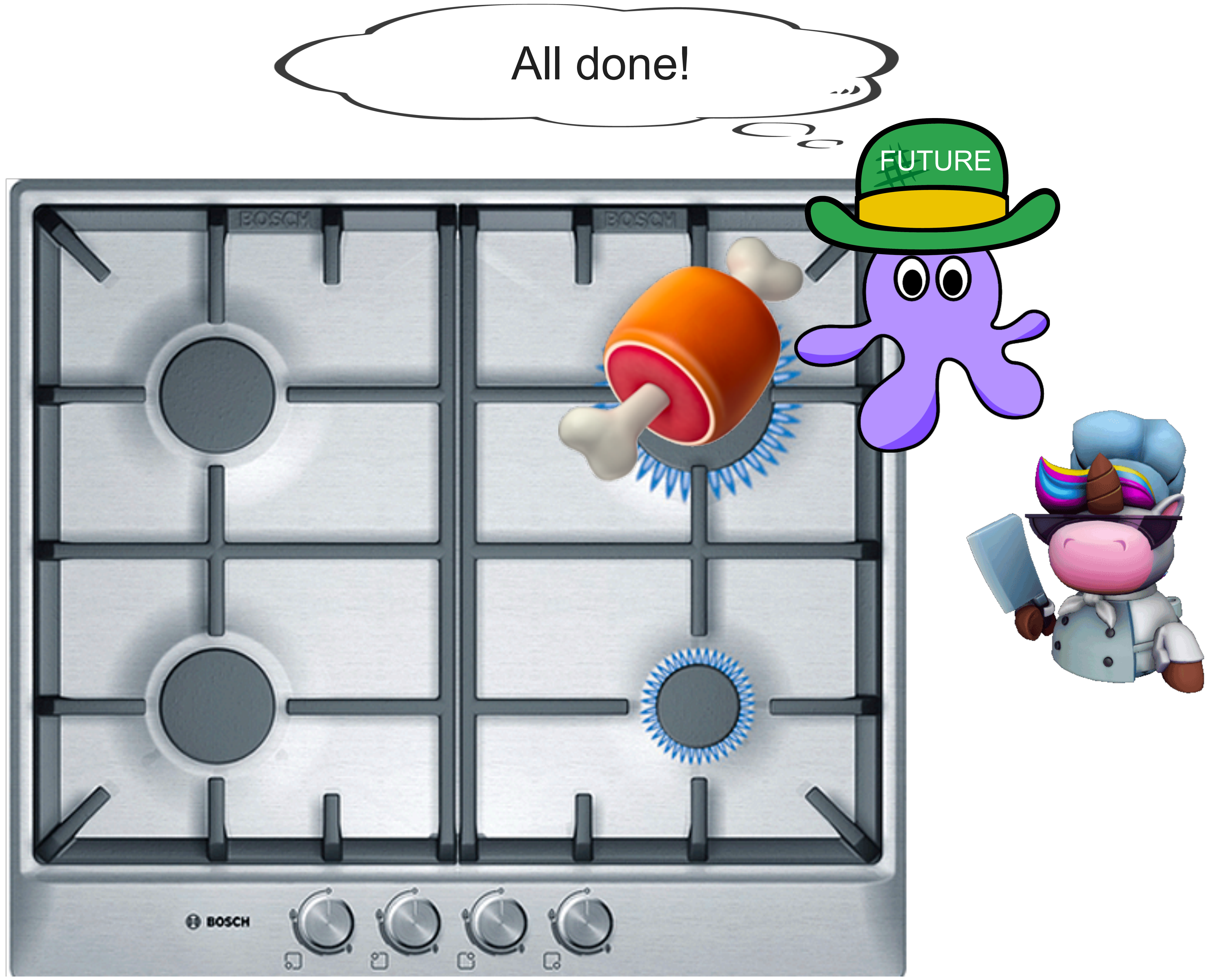
Futures Visualized



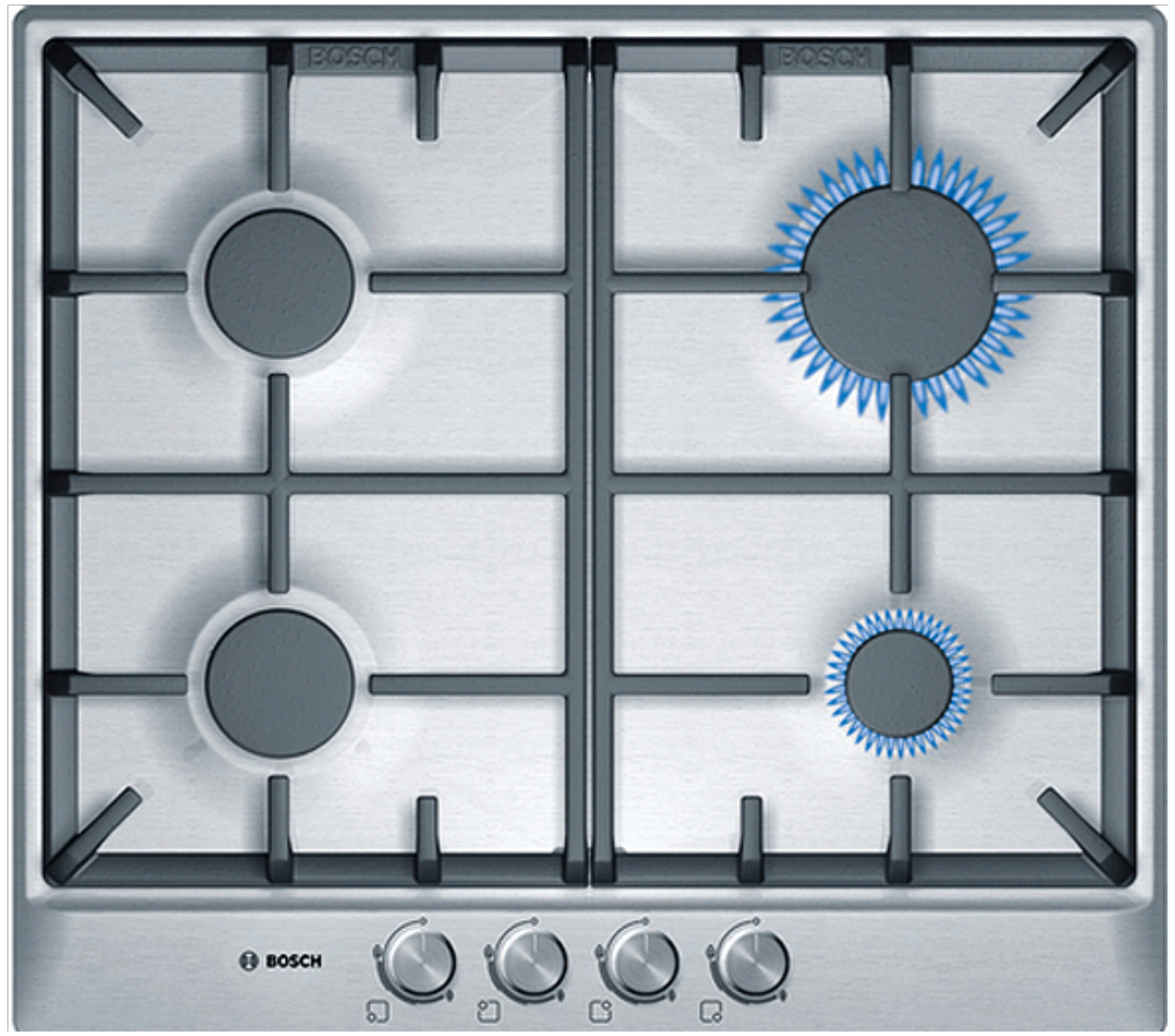
Futures Visualized

Futures Visualized





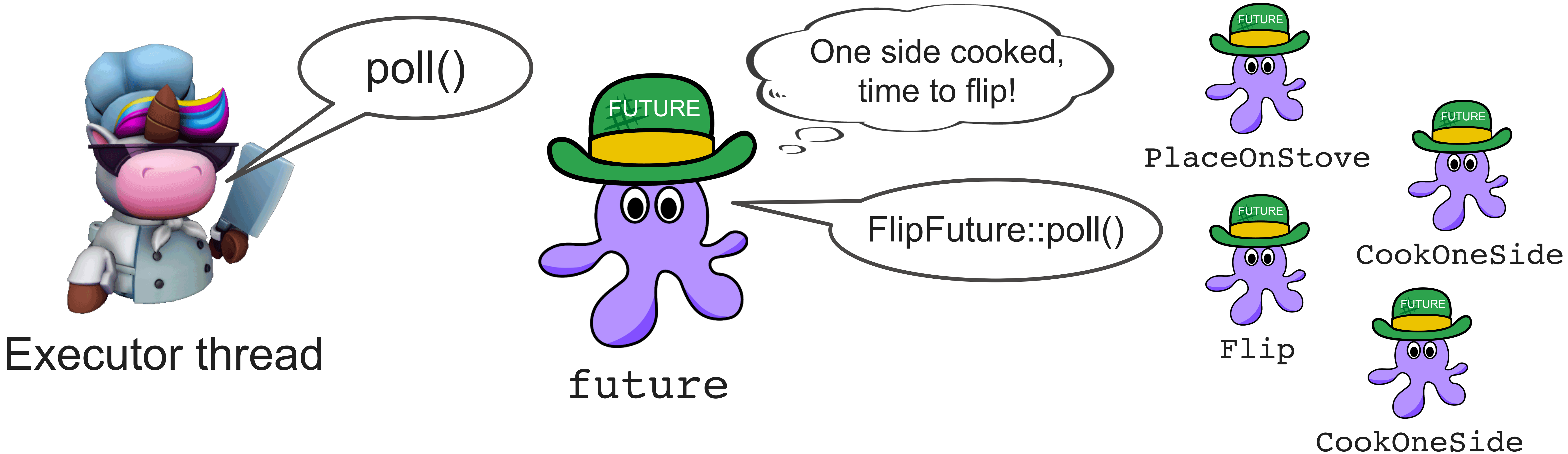
Futures Visualized

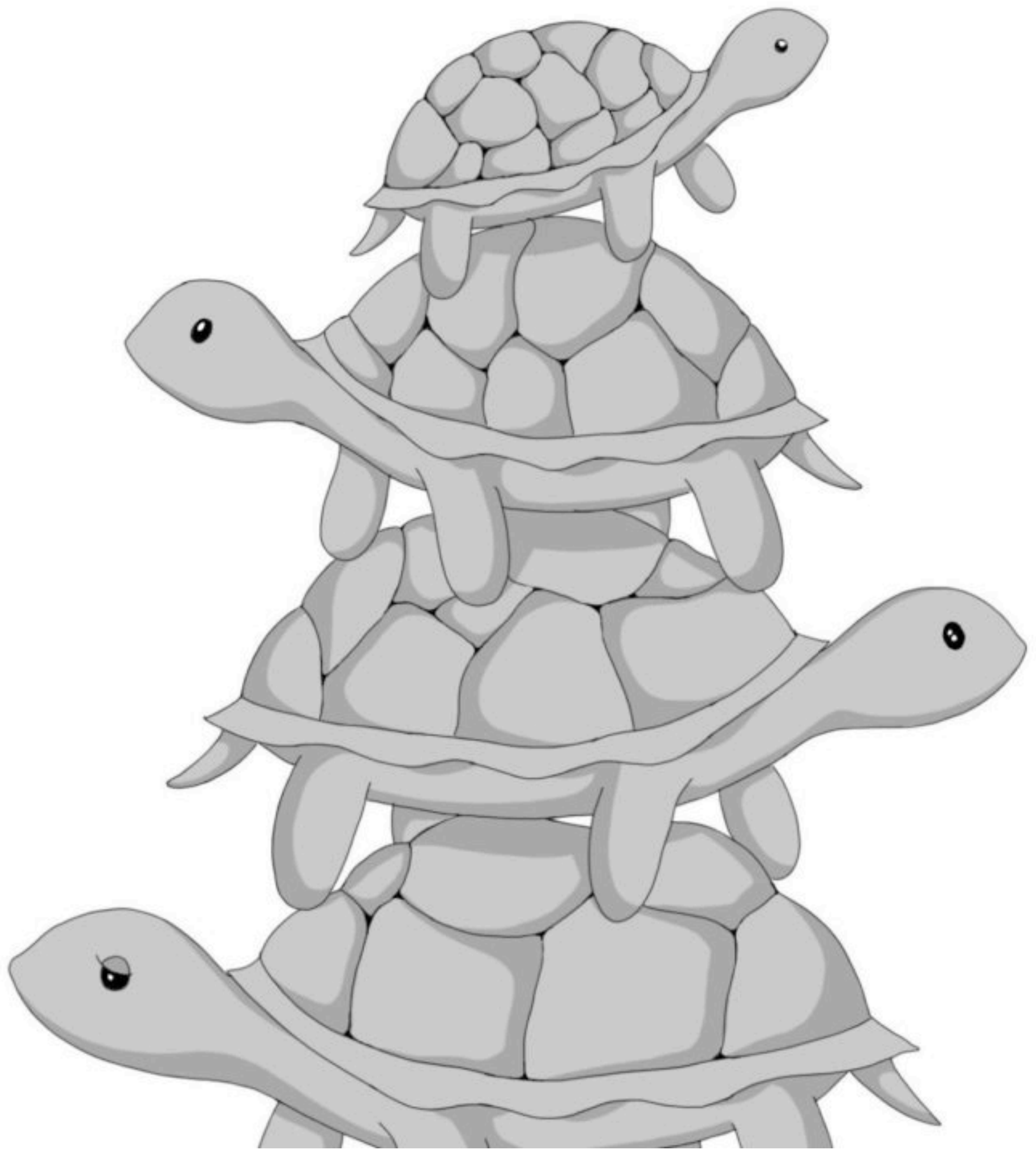


Composition with futures

- Pretty much no one implements futures manually (unless you're a low level library implementor)
- Instead, futures are composed with various [combinators]

```
let future = placeOnStove(meat)
    .then(|meat| cookOneSide(meat))
    .then(|meat| flip(meat))
    .then(|meat| cookOneSide(meat));
```





Ergonomics of futures

Working with futures isn't terribly ergonomic

```
let future = placeOnStove(meat)
    .then(|meat| cookOneSide(meat))
    .then(|meat| flip(meat))
    .then(|meat| cookOneSide(meat));
```

- This code works
    - It's certainly much better than manually dealing with callbacks and state machines as you would in C/C++ with interfaces like epoll!
- But can we do better?
    - The syntax is a little clunky… It's more typing than we'd like
    - Code quickly becomes much messier as complexity increases
    - Sharing mutable data (e.g. in local variables) can be painful: if there can only be one mutable reference at a time, only one closure can touch that data!

Poor ergonomics example

```
fn addToInbox(email_id: u64, recipient_id: u64) -> impl Future<Output=Result<(), Error>> {
    loadMessage(email_id)
        .and_then(|message| get_recipient(message, recipient_id))
        .map(|(message, recipient)| recipient.verifyHasSpace(&message))
        .and_then(|(message, recipient)| recipient.addToInbox(message))
}
```

Asynchronous functions returning Futures

Synchronous (normal) function

Poor ergonomics example

```
fn addToInbox(email_id: u64, recipient_id: u64) -> impl Future<Output=Result<(), Error>> {
    loadMessage(email_id)
        .and_then(|message| get_recipient(message, recipient_id))
        .map(|(message, recipient)| recipient.verifyHasSpace(&message))
        .and_then(|(message, recipient)| recipient.addToInbox(message))
}
```

That’s a mouthful!

Poor ergonomics example

```
fn addToInbox(email_id: u64, recipient_id: u64) -> impl Future<Output=Result<(), Error>> {
    loadMessage(email_id)
        .and_then(|message| get_recipient(message, recipient_id))
        .map(|(message, recipient)| recipient.verifyHasSpace(&message))
        .and_then(|(message, recipient)| recipient.addToInbox(message))
}
```

Strange decomposition: why does get_recipient need to take a Message?

(It doesn't, but we need to pass it in order to make this chain of futures work, since the next futures need both the message and recipient as input. This is bad abstraction!)

Improved ergonomics with syntactic sugar

```
fn addToInbox(email_id: u64, recipient_id: u64)
    -> impl Future<Output=Result<(), Error>>
{
    loadMessage(email_id)
        .and_then(|message|
            get_recipient(message, recipient_id))
        .map(|(message, recipient)|
            recipient.verifyHasSpace(&message))
        .and_then(|(message, recipient)|
            recipient.addToInbox(message))
}
```

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;

    let recipient = get_recipient(recipient_id).await?;

    recipient.verifyHasSpace(&message)?;

    recipient.addToInbox(message).await
}
```

no wonky decomposition!

normal function usage!

- An async function is a function that returns a Future. (Any Futures used in the function are chained together by the compiler.)
- `.await` waits for a future and gets its value
  - `.await` can only be called in an async fn or block
- Everything else is pretty much the same as what you're used to!
- The compiler transforms this code into a Future with a poll() method that is just as efficient as what you could implement by hand!

Asynchronous programming is now really accessible!

- Simple synchronous, threaded echo server:

```rust
use std::io::{Read, Write};
use std::net::TcpListener;
use std::thread;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").unwrap();

    loop {
        let (mut socket, _) = listener.accept().unwrap();

        thread::spawn(move || {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).unwrap();
            socket.write_all(&buf[0..n]).unwrap();
        });
    }
}
```

Asynchronous programming is now really accessible!

- Convert any blocking functions to asynchronous versions (i.e. versions that return Futures)

```
use std::io::{Read, Write};
use std::net::TcpListener;
use std::thread;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080")
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().unwrap();


        thread::spawn(move || {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).unwrap();
            socket.write_all(&buf[0..n]).unwrap();
        });
    }
}
```

```
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080")
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().unwrap();


        tokio::spawn(move || {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).unwrap();
            socket.write_all(&buf[0..n]).unwrap();
        });
    }
}
```

Asynchronous programming is now really accessible!

- Now we have futures — need to .await them!
  - The compiler will complain if you forget

```
use std::io::{Read, Write};
use std::net::TcpListener;
use std::thread;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080")
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().unwrap();


        thread::spawn(move || {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).unwrap();
            socket.write_all(&buf[0..n]).unwrap();
        });
    }
}
```

```
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;


fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").await
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().await
            .unwrap();

        tokio::spawn(move || {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).await.unwrap();
            socket.write_all(&buf[0..n]).await.unwrap();
        });
    }
}
```

Asynchronous programming is now really accessible!

- You can only use .await in an async function or block
  - Compiler will also complain if you forget

```
use std::io::{Read, Write};
use std::net::TcpListener;
use std::thread;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080")
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().unwrap();


        thread::spawn(move || {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).unwrap();
            socket.write_all(&buf[0..n]).unwrap();
        });
    }
}
```

```
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").await
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().await
            .unwrap();

        tokio::spawn(async move {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).await.unwrap();
            socket.write_all(&buf[0..n]).await.unwrap();
        });
    }
}
```

Asynchronous programming is now really accessible!

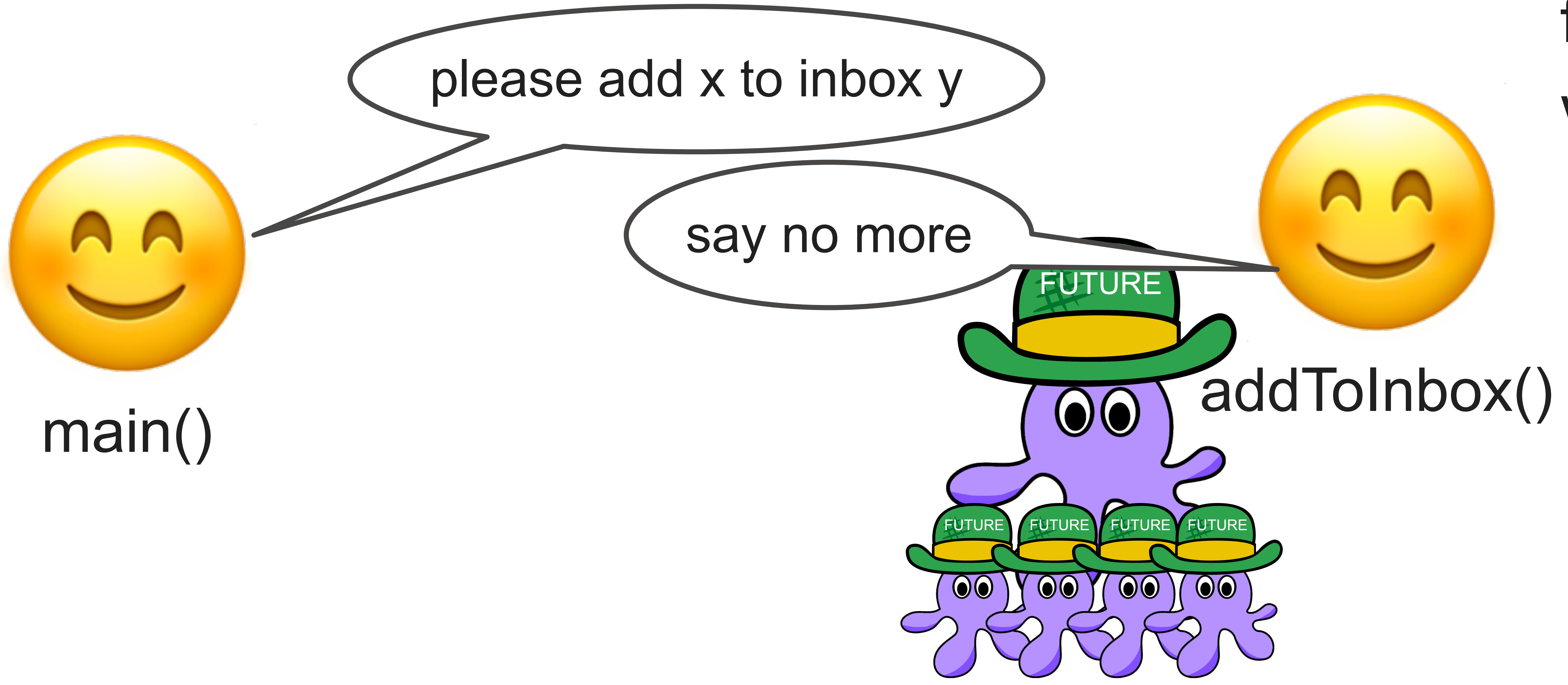
- main() now returns a Future.
  - That's fine, but Futures don't actually do anything unless an executor executes them. Need to run main() and submit the returned Future to the executor!
  - #[tokio::main] is a convenience macro that does this

```
use std::io::{Read, Write};
use std::net::TcpListener;
use std::thread;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080")
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().unwrap();


        thread::spawn(move || {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).unwrap();
            socket.write_all(&buf[0..n]).unwrap();
        });
    }
}
```

```
use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:8080").await
        .unwrap();

    loop {
        let (mut socket, _) = listener.accept().await
            .unwrap();

        tokio::spawn(async move {
            let mut buf = [0; 1024];
            let n = socket.read(&mut buf).await.unwrap();
            socket.write_all(&buf[0..n]).await.unwrap();
        });
    }
}
```

Async functions generate/return futures

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```



- If you run this function, it will not actually do any work with any messages!!
- This is still a function and you can still run it…
- But its purpose is now to produce a future that does the stuff that was written inside the function

Async functions generate/return futures

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

- If you run this function, it will not actually do any work with any messages!!
- This is still a function and you can still run it…
- But its purpose is now to produce a future that does the stuff that was written inside the function

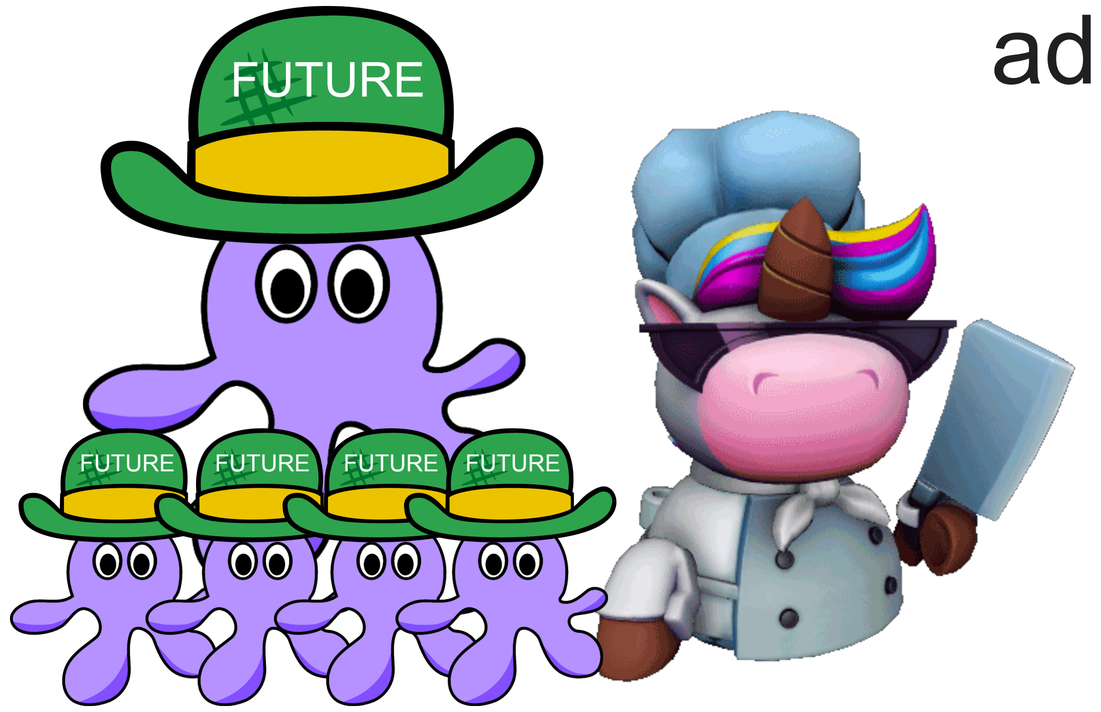
Async functions generate/return futures

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

- If you run this function, it will not actually do any work with any messages!!
- This is still a function and you can still run it…
- But its purpose is now to produce a future that does the stuff that was written inside the function

main()

addToInbox()



Executor thread

Now the email is added to the inbox

How it compiles

- Async/await code looks similar to normal synchronous code, but…
- It’s completely different under the hood!

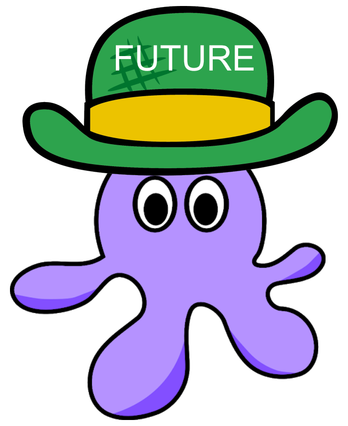
```
fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id)?;
    let recipient = get_recipient(recipient_id)?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message)
}
```

Normal, synchronous code stores variables on the stack
If we need to wait, the OS switches to a different thread with a different stack

addToInbox
get_recipient
make_db_request

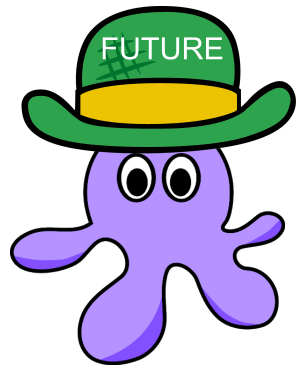
```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

Asynchronous functions return a Future. Any state for the future must be self contained in the future object
… so there isn’t a stack?



How it compiles

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

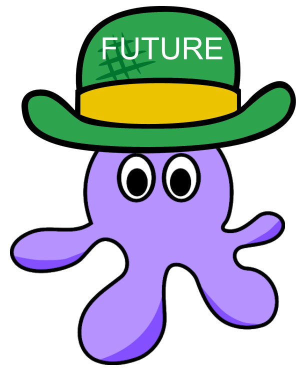



```
enum AddToInboxState {
    NotYetStarted { email_id: u64, recipient_id: u64 },
    WaitingLoadMessage {
        recipient_id: u64, state: LoadMessageFuture },
    WaitingGetRecipient {
        message: Message, state: GetRecipientFuture },
    WaitingAddToInbox {
        state: AddToInboxFuture },
    Completed { result: Result<(), Error> },
}
```

- Looking at this code, there are 5 places where we might be paused, not actively executing:
  - Before anything has happened yet (i.e. Future has been created but not yet poll()ed)
  - await-ing for loadMessage
  - await-ing for get_recipient
  - await-ing for addToInbox
  - Future has completed
- We can use an enum to store the state for these possibilities!
  - An enum compiles like a union type in C: its size is equal to the largest size of its variants. Maximally efficient in storage

How it compiles

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

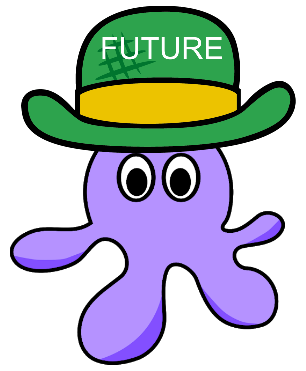



```
enum AddToInboxState {
    NotYetStarted { email_id: u64, recipient_id: u64 },
    WaitingLoadMessage {
        recipient_id: u64, state: LoadMessageFuture },
    WaitingGetRecipient {
        message: Message, state: GetRecipientFuture },
    WaitingAddToInbox {
        state: AddToInboxFuture },
    Completed { result: Result<(), Error> },
}
```

- How should we implement poll() for this Future? We can look at the current state and execute the appropriate code from our async fn

How it compiles

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```




```
enum AddToInboxState {
    NotYetStarted { email_id: u64, recipient_id: u64 },
    WaitingLoadMessage {
        recipient_id: u64, state: LoadMessageFuture },
    WaitingGetRecipient {
        message: Message, state: GetRecipientFuture },
    WaitingAddToInbox {
        state: AddToInboxFuture },
    Completed { result: Result<(), Error> },
}
```

```
fn poll() {
    match self.state {
        NotYetStarted(email_id, recipient_id) => {
            let next_future = load_message(email_id);
            switch to WaitingLoadMessage state
        },
        WaitingLoadMessage(email_id, recipient_id, state) => {
            match state.poll() {
                Ready(message) => {
                    let next_future = get_recipient(recipient_id);
                    switch to WaitingGetRecipient state
                },
                Pending => return Pending,
            }
        },
        WaitingGetRecipient(message, recipient_id, state) => {
            match state.poll() {
                Ready(recipient) => {
                    recipient.verifyHasSpace(&message)?;
                    let next_future = recipient.addToInbox(message);
                    switch to WaitingAddToInbox state
                },
                Pending => return Pending,
            }
        },
        ...
```

**Note: this poll() function is NOT how futures are actually implemented, but it is conceptually how things work. Futures are implemented in terms of a feature called a *generator*; see here or here for more detailed explanation.

Implications

- Async functions have no stack! (sometimes called “stackless coroutines”)
  - The executor thread still has a stack (used to run normal/synchronous functions), but it isn’t used to store state when switching between async tasks. All state is self contained in the generated Future
  - This makes debugging extremely wonky in many languages — how do you get a stack trace if there is no stack?
    - Fortunately, with Rust’s nested futures, it isn’t hard; see here for details
- No recursion!
  - The Future returned by an async function needs to have a fixed size known at compile time
- Rust async functions are nearly optimal in terms of memory usage and allocations
  - There is extremely little overhead. The performance is as good as (or possibly better) what you could get tuning everything by hand

When should I write async code?

- Taking a step back: What were the original problems we were trying to solve with threads?
  - Memory usage from having so many stacks
  - Unnecessary context switching cost
- Async code makes sense when…
  - You need an extremely high degree of concurrency
    - Not as much reason to use async if you don't have that many threads
  - Work is primarily I/O bound
    - Context switching overhead is expensive only if you're using a tiny fraction of the time slice
    - If you're doing a lot of work on the CPU for an extended period of time, you might prevent the executor from running other tasks

Similar tools in other languages

- Rust lets us write asynchronous code in the synchronous style that we're used to. This is becoming more common in many other languages
- Javascript: very similar toolbox with Promises and async/await. Involves much more dynamic memory allocation, not as efficient
- Golang: "goroutines" are the asynchronous tasks, but unlike Rust they are not stackless
  - They have resizable stacks. Possible because Go is garbage collected, so the runtime knows where all pointers are and can reallocate memory
- C++20 just got stackless coroutines! Still lots of sharp edges, may want to wait for more libraries to make this easier to use

General Tips for Async Rust

- Never block in async code!
  - Asynchronous tasks are cooperative (not preemptive)
- You can only use `await` in `async` functions.
- Rust won't let you write async functions in traits (for technical reasons that have to do with lifetimes and the fact that you can't have associated type bounds *yet*)
  - You can use a crate called async-trait though!

Additional Resources/References

- A great talk, high-level overview about how Rust arrived on the design for futures
- A great talk about how futures are implemented, how async/await works under the hood
- A blog post about how async/await is implemented
- Phil Levis' CS110 Lecture on Events, Threads, and Async I/O
- The Rust Docs on Futures
- An article on futures
- John Ousterhout on why threads are a bad idea
- A great (and very accessible) Medium article explaining epoll (also has great illustrations!)
- A CS242 Assignment on Implementing Futures
- Note: the syntax for futures has changed over time so some of these articles may use outdated syntax — for the most up-to-date syntax, check out the docs.