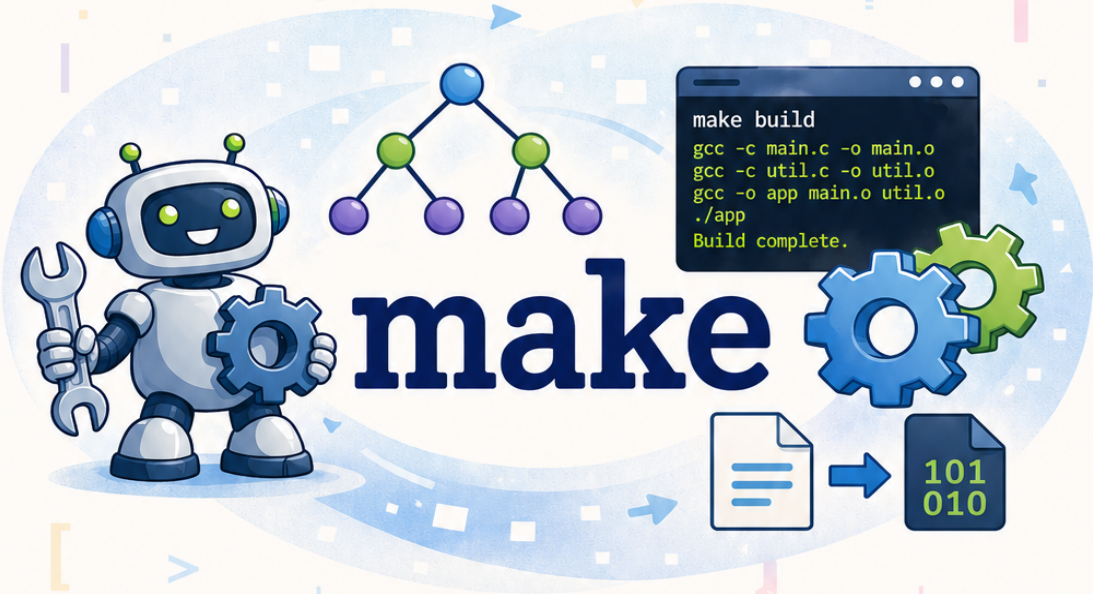
URADICAL

POCKET REFERENCE



Make Pocket Reference

Alan Bradley

Make Pocket Reference

A quick reference guide for dependency-aware task running

Alan Bradley

uradical.io

Make Pocket Reference

Alan Bradley

Table of Contents

Make Is Already Installed

Every developer has a build tool problem. Not the absence of one — the opposite. Task runners, script wrappers, plugin systems, YAML-driven orchestrators. Each one adds a dependency, a version constraint, a thing that can break before your actual work begins.

Make was written in 1976. It ships with every Unix system. It has no runtime, no package manager, no configuration format. You write targets and shell commands. Make runs them in the right order. It has been doing this, reliably, for nearly fifty years.

That's not a limitation. That's a moat.

This guide treats Make as what it actually is in modern practice: a dependency-aware task runner. Not a C build system. Not a relic. The tool that's already on your machine, waiting to replace a dozen scripts and a README full of instructions nobody reads.

The Starter Makefile

Before anything else, here's a complete Makefile you can drop into a project and start editing. Everything in it is explained in the sections that follow.

```
SHELL := /bin/bash
.SHELLFLAGS := -eu -o pipefail -c
.DELETE_ON_ERROR:

BINARY   := server
PKG      := ./cmd/server
BUILD_DIR := bin
VERSION  := $(shell git describe --tags --always --dirty
2>/dev/null || echo dev)
LDFLAGS  := -s -w -X main.version=$(VERSION)

.DEFAULT_GOAL := help

.PHONY: help build test lint clean run docker-build

help: ## Show available targets
    @grep -E '^[a-zA-Z_-]+:.*?## .*$$' $(MAKEFILE_LIST) | \
        awk 'BEGIN {FS = ":.*?## "}; {printf "
\033[36m%-20s\033[0m %s\n", $$1, $$2}'

build: ## Build the binary
    CGO_ENABLED=0 go build -ldflags '$(LDFLAGS)' -o
$(BUILD_DIR)/$(BINARY) $(PKG)

test: ## Run tests
    go test -race -count=1 ./...

lint: ## Run linter
    golangci-lint run ./...

run: build ## Build and run
    ./$(BUILD_DIR)/$(BINARY)

clean: ## Remove build artefacts
    rm -rf $(BUILD_DIR)
```

```
docker-build: ## Build Docker image
    docker build -t $(BINARY):$(VERSION) .
```

Run `make` with no arguments and you get:

```
  help                 Show available targets
  build                Build the binary
  test                 Run tests
  lint                 Run linter
  run                  Build and run
  clean                Remove build artefacts
  docker-build         Build Docker image
```

That self-documenting help target replaces your README's "how to build" section. Every target annotated with `## comment` shows up automatically.

The first four lines are worth pausing on. `SHELL` and `.SHELLFLAGS` enable bash strict mode — commands fail fast on errors instead of silently succeeding. `.DELETE_ON_ERROR` cleans up partial output when a command fails. These three lines prevent an entire category of "works on my machine" bugs. Put them in every Makefile you write.

Now let's understand why all of this works.

Rules: Target, Prerequisites, Commands

A Makefile is a set of **rules**:

```
target: prerequisites
    command
```

The **target** is the thing you want. The **prerequisites** are things that must exist first. The **commands** are what Make runs to produce the target.

Try it. Create a file called `Makefile` with this content:

```
greet:
    echo "Hello from Make"
```

Run `make`:

```
$ make
echo "Hello from Make"
Hello from Make
```

Make prints the command, then its output. That's a complete Makefile. One target, one command, no prerequisites. When you type `make` with no arguments, Make runs the **first target in the file**. That's the default, and it's the only rule for target selection you need to remember. (The starter Makefile above overrides this with `.DEFAULT_GOAL := help`, but the principle is the same — Make needs to know which target to start with, and "first one" is the factory setting.)

Now let's add a prerequisite. Replace the contents with:

```
output.txt: input.txt
    cp input.txt output.txt
    echo "Copied."

input.txt:
    echo "some data" > input.txt
```

Run `make`:

```
$ make
echo "some data" > input.txt
cp input.txt output.txt
Copied.
```

Make sees that `output.txt` needs `input.txt`, that `input.txt` doesn't exist, so it creates it first, then copies it. Run `make` again:

```
$ make
make: 'output.txt' is up to date.
```

Nothing happened. `output.txt` is newer than `input.txt`, so Make knows there's nothing to do. Now edit `input.txt` and run `make` again — it re-runs the copy because the prerequisite is newer than the target.

That timestamp check is the entire engine. Everything else in Make is built on top of it.

The tab rule. Commands must be indented with a literal TAB character, not spaces. If you see `*** missing separator. Stop.`, you have spaces. Configure your editor to insert tabs in Makefiles — most have a filetype setting for this.

Chaining rules

Targets can depend on other targets:

```
deploy: build docker-push
    kubectl apply -f k8s/

docker-push: docker-build
    docker push myapp:latest

docker-build: build
    docker build -t myapp:latest .

build:
```

```
    go build -o bin/server ./cmd/server
```

`make deploy` triggers the entire chain in the right order: build . docker-build . docker-push . deploy. You declare what depends on what. Make figures out the sequence.

.PHONY: This Target Is Not a File

Make assumes every target is a filename. When your target is a verb — `build`, `test`, `clean`, `deploy` — you need to tell Make it's not a file:

```
.PHONY: build test clean deploy
```

Without this, if someone creates a file called `test` in your project root, `make test` silently does nothing. It checks the timestamp, finds the file exists, and declares the target up to date.

Declare all your phony targets in a single line at the top. It's a one-second habit that prevents a baffling five-minute debug session.

Variables

Variables are strings. Define them at the top of the file, reference them with `$()`:

```
BINARY   := server
BUILD_DIR := bin

build:
    go build -o $(BUILD_DIR)/$(BINARY) ./cmd/$(BINARY)
```

:= vs = — use :=

Make has two flavours of variable assignment:

```
# Simply expanded — evaluated right now
VERSION := $(shell git describe --tags 2>/dev/null || echo
dev)

# Recursively expanded — evaluated every time it's referenced
VERSION = $(shell git describe --tags 2>/dev/null || echo dev)
```

With `:=`, Make runs the `git` command once and stores the result. With `=`, Make runs it every single time `$(VERSION)` appears. That's slower, surprising, and occasionally produces different results mid-build.

Default to `:=`. Use `=` only when you intentionally want lazy evaluation and understand the consequences.

?= and +=

```
# Set only if not already defined — useful for defaults
DB_URL ?= postgres://localhost:5432/myapp?sslmode=disable

# Append
LDFLAGS += -X main.version=$(VERSION)
```

Command-line overrides

Any variable can be overridden from the command line:

```
make build VERSION=1.2.3
```

This is how you pass values into a Makefile without editing it. CI pipelines use this constantly.

Automatic Variables

Inside a command, Make gives you shorthand for the current target and its prerequisites. You need four:

```
bin/server: cmd/server/main.go internal/app.go
    # $@  = bin/server          (the target)
    # $<  = cmd/server/main.go  (first prerequisite)
    # $^  = cmd/server/main.go internal/app.go  (all
prerequisites)
    # $?  = whichever of the above are newer than the target
    go build -o $@ ./cmd/server
```

There are others. You don't need them.

Pattern Rules

When you have multiple targets that follow the same shape, pattern rules let you write the rule once:

```
bin/%: cmd/%/main.go
    go build -o $@ ./cmd/$*
```

The `%` is a wildcard. It matches any string — the **stem**. `$*` gives you the stem inside the command.

So `make bin/api` matches with stem `api`, checks for `cmd/api/main.go`, and runs `go build -o bin/api ./cmd/api`.

This is how you build a multi-service monorepo without copy-pasting targets:

```
SERVICES := api worker scheduler

.PHONY: build-all $(addprefix build-,$(SERVICES))

build-all: $(addprefix build-,$(SERVICES)) ## Build all
services

build-%:
    go build -o bin/$* ./cmd/$*
```

`make build-all` builds every service. `make build-api` builds just one. The pattern rule handles the routing.

Commands and the Shell

Each line is a separate shell

This is Make's most common gotcha:

```
# BROKEN — the cd has no effect on the echo
deploy:
    cd /opt/app
    echo "I'm in $$(pwd)"    # Still in the original directory

# FIXED — chain commands on one line
deploy:
    cd /opt/app && echo "I'm in $$(pwd)"
```

Each line of a command runs in its own shell process. Variables set on one line don't exist on the next. Directories changed on one line are forgotten. Chain related commands with `&&` or use backslash continuation:

```
deploy:
    cd /opt/app && \
        source .env && \
        ./run.sh
```

@ silences, - ignores errors

```
test:
    @echo "Running tests..."    # @ stops Make from printing
the command itself
    @go test ./...

clean:
    -rm -rf bin/                # - continues even if rm fails
(e.g., dir doesn't exist)
```

Shell variables need $$

Make owns the `$` character. To pass a dollar sign through to the shell, double it:

```
list-pids:
    @ps aux | awk '{print $$2}'

loop:
    @for f in *.go; do echo $$f; done
```

This is the "why isn't my shell script working in a Makefile" answer 90% of the time.

Functions Worth Knowing

Make has a library of text processing functions. Most are niche. These are the ones that earn their place:

$(shell ...) — run a shell command, capture the output:

```
GIT_SHA  := $(shell git rev-parse --short HEAD)
GO_FILES := $(shell find . -name '*.go' -not -path
'./vendor/*')
```

$(wildcard ...) — glob files safely:

```
SOURCES := $(wildcard cmd/*/main.go)
```

Never use bare `*` in a variable definition. `FILES := *.go` doesn't expand — it stores the literal string `*.go`. Always wrap it in `$(wildcard)`.

$(patsubst pattern,replacement,text) — transform a list:

```
SOURCES  := cmd/api/main.go cmd/worker/main.go
BINARIES := $(patsubst cmd/%/main.go,bin/%,$(SOURCES))
# bin/api bin/worker
```

Shorthand: `$(SOURCES:.go=.o)` replaces suffixes.

$(addprefix prefix,names) — prepend to every item:

```
SERVICES := api worker scheduler
BUILD_TARGETS := $(addprefix build-,$(SERVICES))
# build-api build-worker build-scheduler
```

$(filter pattern,text) and **$(filter-out pattern,text)** — select or reject:

```
FILES   := main.go utils.go main_test.go
SOURCES := $(filter-out %_test.go,$(FILES))
```

$(subst from,to,text) — simple string replacement:

```
VERSION := v1.2.3
CLEAN   := $(subst v,,$(VERSION))
# 1.2.3
```

$(error ...), **$(warning ...)**, **$(info ...)** — communicate:

```
ifndef API_KEY
    $(error API_KEY is required — set it in .env or pass it on
the command line)
endif
```

`$(error)` halts the build with a message. Use it for hard requirements. `$(warning)` prints but continues. `$(info)` is purely informational.

Conditionals

Make evaluates conditionals at parse time, not at runtime:

```
ifeq ($(CI),true)
    LDFLAGS += -s -w
endif

ifdef VERBOSE
    GO_TEST_FLAGS := -v
endif
```

The four directives: `ifeq`, `ifneq`, `ifdef`, `ifndef`.

One subtlety: `ifdef` checks whether a variable has been defined, not whether it's non-empty. A variable set to an empty string passes `ifdef`. To check for empty, use:

```
ifeq ($(strip $(MY_VAR)),)
    $(error MY_VAR is empty)
endif
```

Includes and Local Overrides

Split a large Makefile into modules:

```
include docker.mk
include deploy.mk
```

The `-include` variant (with the dash) silently skips missing files:

```
-include .env.mk
```

This is the pattern for local developer overrides. Create a `.env.mk` with your personal settings, add it to `.gitignore`, and `-include` it from the main Makefile. Everyone gets their own config without cluttering the shared file.

Recursive Make

When your Makefile needs to call Make in a subdirectory, use `$(MAKE)`, not `make`:

```
deploy:
    cd infrastructure && $(MAKE) apply
```

`$(MAKE)` passes through flags like `-j` (parallel jobs) and `-k` (keep going). Bare `make` drops them silently, producing different behaviour in CI than locally.

Parallel Execution

By default, Make runs one command at a time. Add `-j` and it runs independent targets simultaneously:

```
make -j4 build-all
```

This tells Make to run up to four commands in parallel. On a multi-service repo where `build-api`, `build-worker`, and `build-scheduler` don't depend on each other, `-j` builds all three at once. On a CI machine, `make -j$(nproc)` uses every available core.

The critical thing to understand: **Make's only protection against race conditions is the dependency graph you wrote.** If two targets don't have an explicit prerequisite relationship, `-j` assumes they can run at the same time. If those targets both write to the same directory, both read from a shared config that one of them modifies, or both use a tool that doesn't handle concurrent invocations, you have a race condition.

```
# SAFE under -j — test and lint are independent
.PHONY: check test lint

check: test lint

test:
    go test ./...

lint:
    golangci-lint run ./...
```

```
# UNSAFE under -j — both targets write to the same output file
.PHONY: report check-api check-worker

report: check-api check-worker

check-api:
    run-checks api >> results.txt

check-worker:
```

```
    run-checks worker >> results.txt
```

The fix is either to give each target its own output, or to add a dependency so they run in sequence:

```
# FIXED — each target writes to its own file, then report
combines them
check-api:
    run-checks api > results-api.txt

check-worker:
    run-checks worker > results-worker.txt

report: check-api check-worker
    cat results-*.txt > results.txt
```

A useful habit: run `make -j -n` (parallel dry run) on any new Makefile. If the printed command order looks wrong — a deploy running before a build, a push running before a tag — you're missing a prerequisite. Fix the graph before you run it for real.

Patterns You Can Steal

The examples below are complete — copy them into your project and edit to fit. Each one demonstrates a specific technique from earlier in the guide working in a real context.

Environment-specific configuration

Most projects deploy to more than one environment. Rather than littering your Makefile with conditionals, use includes and variable overrides to keep each environment's config in its own file:

```
ENV ?= development

-include config/$(ENV).mk

deploy: ## Deploy to current environment
    @echo "Deploying to $(ENV)..."
    @echo "  Host: $(DEPLOY_HOST)"
    @echo "  Path: $(DEPLOY_PATH)"
    ssh $(DEPLOY_HOST) "cd $(DEPLOY_PATH) && git pull && make
build"
```

With `config/staging.mk`:

```
DEPLOY_HOST := staging.example.com
DEPLOY_PATH := /opt/myapp
```

And `config/production.mk`:

```
DEPLOY_HOST := prod.example.com
DEPLOY_PATH := /opt/myapp
```

Then `make deploy ENV=staging` does what you'd expect. This combines three things from earlier — `?=` defaults, `-include` for optional files, and command-line overrides — into one clean pattern. The config files aren't committed to the Makefile itself, so adding a new environment means adding a new `.mk` file, not editing shared

infrastructure.

Docker with tagging

Docker builds have two moving parts: the image tag and the registry. Pin both to git state and your builds become reproducible without manual version bumping:

```
IMAGE    := myapp
REGISTRY := ghcr.io/myorg
TAG      := $(shell git rev-parse --short HEAD)

.PHONY: docker-build docker-push

docker-build: ## Build and tag Docker image
    docker build -t $(REGISTRY)/$(IMAGE):$(TAG) .
    docker tag $(REGISTRY)/$(IMAGE):$(TAG)
$(REGISTRY)/$(IMAGE):latest

docker-push: docker-build ## Push to registry
    docker push $(REGISTRY)/$(IMAGE):$(TAG)
    docker push $(REGISTRY)/$(IMAGE):latest
```

The prerequisite on `docker-push` means you can't push without building. Make enforces the ordering that your CI script would otherwise have to spell out manually.

Database migrations

Migration commands are the kind of thing developers look up every time. Wrapping them in Make targets turns tribal knowledge into a discoverable interface:

```
DB_URL ?= postgres://localhost:5432/myapp?sslmode=disable

.PHONY: migrate-up migrate-down migrate-create

migrate-up: ## Run pending migrations
    migrate -path migrations -database '$(DB_URL)' up
```

```
migrate-down: ## Rollback last migration
    migrate -path migrations -database '$(DB_URL)' down 1

migrate-create: ## Create migration (make migrate-create
NAME=add_users)
    migrate create -ext sql -dir migrations -seq $(NAME)
```

`DB_URL` uses `?=` so each developer can override it locally without changing the Makefile. The `migrate-create` target takes a `NAME` variable from the command line — `make migrate-create NAME=add_users` — which is a pattern worth reusing any time a target needs a one-off parameter.

Elixir / Phoenix

Phoenix projects have a multi-step lifecycle — deps, database, assets, release. A Makefile turns "read the deployment docs" into `make release`:

```
MIX_ENV ?= dev

.PHONY: setup test server release

setup: ## Initial project setup
    mix deps.get
    mix ecto.setup
    cd assets && npm install

test: ## Run tests
    MIX_ENV=test mix test

server: ## Start Phoenix dev server
    mix phx.server

release: ## Build production release
    MIX_ENV=prod mix deps.get --only prod
    MIX_ENV=prod mix compile
    MIX_ENV=prod mix assets.deploy
    MIX_ENV=prod mix release
```

Notice `setup` runs three commands across two directories. Since each line runs in its own shell, the `cd assets` on the last line doesn't affect anything — it's self-contained. That's the separate-shell behaviour from earlier working in your favour for once.

Quick Reference Card

Command-line flags

Flag	What it does
-n	See what Make would do without touching anything
-j N	Run up to N commands in parallel — serious speed gains on multi-target builds
-k	Keep going after errors — useful for seeing all failures at once
-s	Silent mode — suppress command echo entirely
-f FILE	Use a specific file instead of Makefile
-B	Ignore timestamps and rebuild everything unconditionally

Automatic variables

Variable	Expands to
$@	The target — what you're building
$<	The first prerequisite — the primary input
$^	All prerequisites, deduplicated — every input
$?	Only the prerequisites that changed — what triggered the rebuild
$*	The stem — whatever % matched in a pattern rule

Special targets

Target	Effect
.PHONY	Declare targets that aren't files — prevents false "up to date"
.DEFAULT_GOAL	Which target runs when you type make with no arguments
.DELETE_ON_ERROR	Clean up the target if a command fails — prevents corrupt output
.SILENT	Suppress command echo for specific targets

Built-in variables

Variable	Contains
$(MAKE)	The make command itself — always use this for recursive calls
$(MAKEFILE_LIST)	Every Makefile being processed — powers the self-documenting help trick
$(CURDIR)	The directory Make was invoked from
$(SHELL)	The shell running your commands — /bin/sh unless you change it

Further Reading

- GNU Make Manual — the authoritative reference when you need the full picture
- Your Makefiles are wrong — the article that convinced a generation of developers to add strict mode to their Makefiles
- Makefile Tutorial by Example — comprehensive and example-driven, excellent for exploring the corners

---

Published by uRadical — uradical.io

Make. Boring technology. Interesting work.