Running Claude Code Autonomously Overnight: A Complete Developer Workflow Guide

Running Claude Code unattended for overnight development is achievable with the right configuration stack: **WSL2 with tmux for session persistence, Docker sandboxing for safety, hooks for automation, and CLAUDE.md for project-specific instructions**. This guide provides production-ready configurations for Python/QuantConnect trading bot development on Windows 10, covering everything from environment setup through monitoring and recovery.

The core challenge is balancing autonomy with safety. Claude Code's `--dangerously-skip-permissions` flag enables unattended operation PromptLayer but requires containerized isolation to prevent unintended system modifications. Combined with pre-configured hooks, automatic commits, and webhook notifications, you can wake up to meaningful progress on your trading algorithms without risking your development environment.

Environment architecture for WSL2 persistent sessions

The foundation of overnight operation is ensuring your WSL2 session survives disconnection and idle timeouts. WSL2 instances terminate **~15 seconds** after all processes exit, GitHub so you need persistent processes running inside the VM.

WSL2 configuration for unattended operation

Create `/etc/wsl.conf` in your Ubuntu distribution to enable systemd and automatic service startup: microsoft

```ini
[boot]
systemd=true
command="service cron start"

[automount]
enabled=true
options="metadata,umask=22,fmask=11"

[user]
default=yourusername
```

On the Windows side, create `%UserProfile%\.wslconfig` to allocate adequate resources for long-running processes: microsoft

```ini
```

```
[wsl2]
memory=8GB
processors=4
swap=4GB
localhostForwarding=true

[experimental]
sparseVhd=true
```

After creating these files, restart WSL with `wsl --shutdown` followed by opening a new terminal. Microsoft Learn microsoft

Terminal persistence with tmux

Tmux maintains your session across disconnections and reboots. himmelwright Install it and configure for development work:

```bash
sudo apt update && sudo apt install -y tmux
```

Create `~/.tmux.conf` with settings optimized for long-running autonomous sessions:

```bash
set -g history-limit 50000
set -g mouse on
set -sg escape-time 0
setw -g mode-keys vi

# Status bar showing session health
set -g status on
set -g status-interval 5
set -g status-position bottom

# Easy pane management
bind | split-window -h -c "#{pane_current_path}"
bind - split-window -v -c "#{pane_current_path}"
bind r source-file ~/.tmux.conf \; display-message "Config reloaded!"

# Don't rename windows automatically
set-option -g allow-rename off
```

Create a startup script at `~/scripts/claude-overnight.sh`:

```bash
#!/bin/bash
SESSION="claude-dev"
PROJECT_DIR="$HOME/projects/quantconnect-bot"

if ! tmux has-session -t $SESSION 2>/dev/null; then
  tmux new-session -d -s $SESSION -n 'claude' -c $PROJECT_DIR
  tmux send-keys -t $SESSION:claude "source .venv/bin/activate" C-m
  tmux new-window -t $SESSION -n 'tests' -c $PROJECT_DIR
  tmux new-window -t $SESSION -n 'git' -c $PROJECT_DIR
  echo "Session created: $SESSION"
fi

tmux attach -t $SESSION
```

VS Code Remote-WSL integration

Install the **Remote - WSL** extension in VS Code, Microsoft Learn then configure settings.json for the WSL environment:

```json
{
  "terminal.integrated.defaultProfile.linux": "bash",
  "terminal.integrated.profiles.linux": {
    "tmux": {
      "path": "/usr/bin/tmux",
      "args": ["new-session", "-A", "-s", "vscode"]
    }
  },
  "files.autoSave": "afterDelay",
  "files.autoSaveDelay": 1000,
  "remote.WSL.useShellEnvironment": true,
  "[python]": {
    "editor.defaultFormatter": "ms-python.black-formatter",
    "editor.formatOnSave": true
  }
}
```

CLAUDE.md architecture for autonomous operation

CLAUDE.md files provide persistent instructions that Claude Code loads automatically at session start. The file hierarchy allows global, project-wide, and local configurations:

Location	Scope	Version Control
`~/.claude/CLAUDE.md`	All projects	Personal, not committed
`<repo>/CLAUDE.md`	Project-wide	Commit to git
`<repo>/CLAUDE.local.md`	Personal project	Add to .gitignore
`<subdirectory>/CLAUDE.md`	Subdirectory-specific	Loaded on demand

Structure for QuantConnect development

Create your project's `CLAUDE.md` with autonomous operation directives:

markdown

```markdown
# QuantConnect Trading Bot - Claude Code Instructions

## Autonomous Operation Directives
IMPORTANT: You are running in autonomous overnight mode.
- Create checkpoint commits before major changes
- Run tests after every code modification
- If tests fail 3 times consecutively, create a failure report and stop
- Use /compact at 70% context usage, not waiting for auto-compact
- Document progress in PROGRESS.md before each /clear

## Build Commands
- `pytest tests/`: Run all unit tests
- `mypy src/ --strict`: Type checking
- `ruff check --fix .`: Lint and auto-fix
- `lean backtest "TradingBot"`: Run LEAN backtester locally

## Code Standards
- Use type hints on all functions
- Follow QuantConnect's algorithm framework patterns
- Prefer composition over inheritance for signal generators
- All strategies must inherit from QCAlgorithm
- Use self.Debug() for logging, not print()

## Project Structure
- src/algorithms/: Trading algorithm implementations
- src/indicators/: Custom technical indicators
- src/signals/: Entry/exit signal generators
- tests/: Pytest unit tests mirroring src structure

## Error Recovery
If you encounter repeated failures:
1. Document the error in ERRORS.md with timestamp
2. Create a checkpoint: git commit -m "checkpoint: before recovery attempt"
3. Try alternative approach
4. If still failing, document findings and await human review

## Testing Requirements
Before any commit:
- All pytest tests must pass
- mypy --strict shows no errors
- ruff check returns clean
```

Use emphasis markers like **"IMPORTANT"** and **"YOU MUST"** to improve adherence to critical instructions. anthropic Keep the file under **100 lines** focused on project-specific patterns that aren't obvious from the codebase. Shuttle

---

Hooks configuration for automation

Hooks are user-defined commands that execute at specific lifecycle points. Configure them in `.claude/settings.json` for your project:

Available hook events

Event	Trigger	Use Cases
`PreToolUse`	Before any tool	Block dangerous commands, security validation
`PostToolUse`	After tool completes	Auto-formatting, logging, notifications
`Stop`	When Claude finishes	Force continuation, session summary
`SessionStart`	Session begins	Load context, environment setup
`PreCompact`	Before compaction	Backup transcript, preserve state

Production hooks configuration

Create `.claude/settings.json` with hooks for autonomous development:

```
json
```

```json
{
 "env": {
  "BASH_DEFAULT_TIMEOUT_MS": "1800000",
  "BASH_MAX_TIMEOUT_MS": "7200000"
 },
 "permissions": {
  "allow": [
   "Read",
   "Write(src/**)",
   "Write(tests/**)",
   "Edit",
   "Bash(pytest *)",
   "Bash(mypy *)",
   "Bash(ruff *)",
   "Bash(git add *)",
   "Bash(git commit *)",
   "Bash(lean backtest *)"
  ],
  "deny": [
   "Bash(rm -rf *)",
   "Bash(sudo:*)",
   "Write(.env*)",
   "Write(*.key)",
   "Bash(curl *)",
   "Bash(wget *)"
  ]
 },
 "hooks": {
  "PostToolUse": [
   {
    "matcher": "Edit|Write",
    "hooks": [
     {
      "type": "command",
      "command": "jq -r '.tool_input.file_path' | { read fp; if echo \"$fp\" | grep -q '\\.py$'; then ruff check --fix \"$fp\" 2>/de
      "timeout": 30
     }
    ]
   }
  ],
  "PreToolUse": [
   {
    "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.command' >> ~/.claude/command-log.txt"
          }
        ]
      }
    ]
  }
}
```

Auto-commit hook pattern

Add a git commit hook that runs tests before allowing commits:

```json
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "jq -r '.tool_input.command' | grep -q 'git commit' && { pytest -q --tb=no || exit 2; }"
          }
        ]
      }
    ]
  }
}
```

The exit code **2** blocks the action, Anthropic preventing commits when tests fail.

Autonomous mode configuration and safety

The `--dangerously-skip-permissions` flag bypasses all permission checks for unattended operation. **Never use this on your host system**—always run inside a container.

Launching autonomous sessions

Create a startup script that combines all safety measures:

```bash
#!/bin/bash
# start-autonomous.sh

PROJECT_DIR="${1:-$(pwd)}"
TASK="${2:-Continue development according to CLAUDE.md}"

# Ensure we're in tmux
if [ -z "$TMUX" ]; then
  echo "Error: Run inside tmux for session persistence"
  exit 1
fi

# Create checkpoint before starting
cd "$PROJECT_DIR"
git add -A && git commit -m "checkpoint: before autonomous session $(date +%Y%m%d_%H%M%S)" || true

# Start Claude Code with permissions bypass in container
docker run --rm -it \
  -v "$PROJECT_DIR":/workspace \
  -v ~/.claude:/home/appuser/.claude:ro \
  -e ANTHROPIC_API_KEY="$ANTHROPIC_API_KEY" \
  --cpus="2" \
  --memory="4g" \
  --network=none \
  claude-sandbox \
  --dangerously-skip-permissions \
  -p "$TASK"
```

Allowlists for permitted operations

Configure granular permissions in `.claude/settings.json`:

```json
```

```
{
 "permissions": {
  "allow": [
   "Read(**)",
   "Write(src/**)",
   "Write(tests/**)",
   "Write(docs/**)",
   "Bash(python *)",
   "Bash(pytest *)",
   "Bash(git status)",
   "Bash(git add *)",
   "Bash(git commit *)",
   "Bash(git diff *)",
   "Bash(lean backtest *)"
  ],
  "deny": [
   "Read(.env*)",
   "Write(.git/**)",
   "Bash(rm -rf *)",
   "Bash(sudo *)",
   "Bash(curl *)",
   "Bash(pip install *)",
   "WebFetch(*)"
  ]
 }
}
```

Rate limit management

Claude Code uses **rolling 5-hour usage windows**. TrueFoundry For overnight runs spanning 8-10 hours, you'll likely encounter limits with Pro plans. Strategies to manage this:

Use the **Max plan** for near-autonomous capacity—Max20 provides 20x the Pro allowance. Monitor context usage with the `/context` command and compact proactively at **70% capacity** rather than waiting for auto-compact at 95%. MCPcat

Configure extended bash timeouts for long-running backtests:

```json
```

```
{
  "env": {
    "BASH_DEFAULT_TIMEOUT_MS": "1800000",
    "BASH_MAX_TIMEOUT_MS": "7200000"
  }
}
```

Self-recovery and retry mechanisms

Autonomous sessions inevitably encounter errors. Build recovery patterns into your workflow through CLAUDE.md instructions and external tooling.

Context window management

Context degradation is the primary cause of session quality decline. The context window holds **200K tokens** (500K for Sonnet 4 on Enterprise). Claude Implement these patterns:

Document and clear strategy: Instruct Claude to dump progress to a file before clearing:

```markdown
## Context Management Protocol
When context reaches 70%:
1. Write current progress to PROGRESS.md
2. Commit changes: git commit -am "progress: [summary]"
3. Execute /compact with instruction: "preserve the algorithm patterns we established"
4. If still above 80% after compact, execute /clear and resume from PROGRESS.md
```

Session continuation: Resume interrupted sessions with:

```bash
# Continue most recent session
claude --continue "Resume from PROGRESS.md, the last task was..."

# Resume specific session by ID
claude --resume <session-id>
```

Community tools for auto-recovery

Several community tools address common failure modes:

CCAutoRenew monitors usage and auto-renews sessions: GitHub

```bash
./claude-daemon-manager.sh start --at "22:00" --message "continue overnight development"
```

claude-auto-resume detects rate limits and resumes when they lift:

```bash
claude-auto-resume "implement the RSI divergence indicator"
```

Checkpoint-based recovery

Claude Code includes built-in checkpoints that save code state before each change. Access recovery with:

- Press `Esc` twice to access rewind menu
- Use `/rewind` command to rollback to a previous state

Combine with git checkpoints for persistent recovery:

```bash
#!/bin/bash
# checkpoint.sh - Create named recovery point

create_checkpoint() {
    local name="${1:-$(date +%Y%m%d_%H%M%S)}"
    git add -A
    git commit -m "checkpoint: $name" || true
    git tag -a "checkpoint-$name" -m "Automated checkpoint"
    echo "✅ Created checkpoint-$name"
}

restore_checkpoint() {
    git stash push -m "Before restoring $1"
    git reset --hard "$1"
    echo "✅ Restored to $1"
}
```

Git automation for AI-generated commits

Automated commits during autonomous work require careful strategy to maintain a useful git history.

Commit conventions for AI work

Use conventional commits format with an AI indicator:

```
feat(trading): add RSI divergence detection

[AI-GENERATED]

- Implemented RSI divergence algorithm for buy/sell signals
- Added unit tests covering edge cases
- Integrated with existing signal framework
```

Pre-commit configuration

Create `.pre-commit-config.yaml` for quality gates:

```yaml
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.14.7
    hooks:
      - id: ruff-check
        args: [--fix]
      - id: ruff-format

  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.16.1
    hooks:
      - id: mypy
        args: [--ignore-missing-imports, --strict]

  - repo: local
    hooks:
      - id: pytest-check
        name: pytest
        entry: pytest --tb=short -q
        language: system
        types: [python]
        pass_filenames: false
```

Automatic rollback on test failure

Create a pre-push hook that rolls back failed changes:

```bash
#!/bin/bash
# .git/hooks/pre-push

if ! pytest --tb=short -q; then
  echo "❌ Tests failed - finding last good commit..."
  LAST_GOOD=$(git describe --tags --match="checkpoint-*" --abbrev=0 2>/dev/null)

  if [ -n "$LAST_GOOD" ]; then
    echo "Rolling back to $LAST_GOOD"
    git reset --hard "$LAST_GOOD"
  fi
  exit 1
fi

# Create checkpoint on successful push
git tag -a "checkpoint-$(date +%Y%m%d-%H%M%S)" -m "Pre-push checkpoint"
exit 0
```

Testing and validation loops

For QuantConnect development, integrate LEAN backtesting into your continuous validation workflow.

LEAN CLI integration

Install and configure the LEAN CLI:

```bash
pip install lean
lean init
lean login -u YOUR_USER_ID -t YOUR_API_TOKEN
```

Create a backtest validation script:

```bash
```

```
#!/bin/bash
# validate-strategy.sh

PROJECT="${1:-TradingBot}"
MIN_SHARPE="${2:-0.5}"

echo "🧪 Running backtest for $PROJECT..."

if lean backtest "$PROJECT" --data-provider-historical Local; then
  # Extract metrics from results
  RESULT_FILE=$(find "$PROJECT/backtests" -name "*.json" -type f | head -1)
  SHARPE=$(jq -r '.Statistics.SharpeRatio // "0"' "$RESULT_FILE")

  if (( $(echo "$SHARPE >= $MIN_SHARPE" | bc -l) )); then
    echo "✅ Backtest passed: Sharpe=$SHARPE"
    exit 0
  else
    echo "⚠️ Sharpe ratio $SHARPE below threshold $MIN_SHARPE"
    exit 1
  fi
else
  echo "❌ Backtest failed"
  exit 1
fi
```

Continuous testing during development

Add testing directives to CLAUDE.md:

```markdown
## Testing Protocol
After every code change to src/:
1. Run: pytest tests/ -x --tb=short
2. Run: mypy src/ --strict
3. If both pass, proceed
4. If either fails, fix before continuing

Before committing:
1. Run full test suite: pytest tests/ -v
2. Run backtest: ./scripts/validate-strategy.sh
3. Only commit if Sharpe ratio >= 0.5
```

Monitoring and logging

Track Claude Code activity and receive notifications when important events occur.

Activity logging configuration

Create a logging hook in `.claude/settings.json`:

json

```json
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "*",
        "hooks": [
          {
            "type": "command",
            "command": "jq -c '{timestamp: now | strftime(\"%Y-%m-%dT%H:%M:%S\"), tool: .tool_name, input: .tool_input}' >
          }
        ]
      }
    ]
  }
}
```

Discord notification script

Create `scripts/notify.py` for real-time alerts:

python

```
#!/usr/bin/env python3
import os
import requests
from datetime import datetime

DISCORD_WEBHOOK = os.getenv("DISCORD_WEBHOOK_URL")

def notify(title: str, message: str, color: int = 0x00FF00):
    if not DISCORD_WEBHOOK:
        return

    requests.post(DISCORD_WEBHOOK, json={
        "embeds": [{
            "title": title,
            "description": message,
            "color": color,
            "timestamp": datetime.utcnow().isoformat()
        }]
    })

def notify_complete(task: str, duration: float, success: bool):
    color = 0x00FF00 if success else 0xFF0000
    status = "✅ Complete" if success else "❌ Failed"
    notify(f"Claude Task {status}", f"**{task}**\nDuration: {duration:.1f}s", color)

def notify_error(error: str):
    notify("⚠️ Claude Error", f"```\n{error[:1500]}\n```", 0xFF0000)
```

Trigger notifications from hooks:

```
json
```

```
{
 "hooks": {
  "Stop": [
   {
    "matcher": "",
    "hooks": [
     {
      "type": "command",
      "command": "python3 ~/scripts/notify.py complete"
     }
    ]
   }
  ]
 }
}
```

Docker sandboxing for safe overnight operation

Running Claude Code with `--dangerously-skip-permissions` requires container isolation to prevent unintended system modifications.

Dockerfile for isolated development

```dockerfile
FROM ubuntu:22.04

RUN groupadd -r appuser && useradd -r -g appuser appuser

RUN apt-get update && apt-get install -y \
    python3 python3-pip python3-venv \
    git curl nodejs npm \
    && rm -rf /var/lib/apt/lists/*

RUN npm install -g @anthropic-ai/claude-code@latest

WORKDIR /workspace
RUN python3 -m venv /home/appuser/.venv
ENV PATH="/home/appuser/.venv/bin:$PATH"

USER appuser
ENTRYPOINT ["claude"]
```

docker-compose.yml with security constraints

yaml

```yaml
version: '3.8'

services:
  claude-agent:
    build: .
    container_name: claude-overnight
    volumes:
      - ./workspace:/workspace
      - ~/.claude:/home/appuser/.claude:ro
    environment:
      - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 4G
    read_only: true
    tmpfs:
      - /tmp
      - /var/tmp
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true
    networks:
      - isolated
    healthcheck:
      test: ["CMD", "pgrep", "-f", "claude"]
      interval: 30s
      timeout: 10s
      retries: 3

networks:
  isolated:
    driver: bridge
    internal: true  # No external network access
```

Watchdog for runaway processes

Create `scripts/watchdog.py` to terminate hung sessions:

python

```python
#!/usr/bin/env python3
import subprocess
import time
from datetime import datetime

MAX_RUNTIME = 3600 * 8  # 8 hours max
CHECK_INTERVAL = 300    # Check every 5 minutes
CONTAINER = "claude-overnight"

def get_uptime():
    result = subprocess.run(
        ["docker", "inspect", "-f", "{{.State.StartedAt}}", CONTAINER],
        capture_output=True, text=True
    )
    if result.returncode != 0:
        return None
    # Parse and calculate uptime
    return 0  # Simplified

def restart_if_needed():
    uptime = get_uptime()
    if uptime and uptime > MAX_RUNTIME:
        print(f"[{datetime.now()}] Restarting after {uptime}s")
        subprocess.run(["docker", "restart", CONTAINER])

while True:
    restart_if_needed()
    time.sleep(CHECK_INTERVAL)
```

Complete project file templates

Directory structure

```
your-trading-bot/
├── .claude/
│   ├── settings.json        # Hooks and permissions
│   ├── settings.local.json  # Personal settings (gitignored)
│   └── commands/            # Custom slash commands
├── CLAUDE.md                # Project instructions
├── CLAUDE.local.md          # Personal notes (gitignored)
├── scripts/
│   ├── start-autonomous.sh    # Launch overnight session
│   ├── checkpoint.sh          # Git checkpoint management
│   ├── notify.py              # Webhook notifications
│   └── validate-strategy.sh   # Backtest validation
├── src/
│   └── algorithms/
├── tests/
├── docker-compose.yml
├── Dockerfile
├── .pre-commit-config.yaml
├── pyproject.toml
└── .gitignore
```

Overnight session launcher

Create `scripts/overnight-session.sh`:

```bash
```

```
#!/bin/bash
set -e

PROJECT_DIR="$(cd "$(dirname "$0")/.." && pwd)"
TASK="${1:-Continue development per CLAUDE.md}"
LOG_FILE="$PROJECT_DIR/logs/overnight-$(date +%Y%m%d).log"

mkdir -p "$PROJECT_DIR/logs"

# Create pre-session checkpoint
cd "$PROJECT_DIR"
git add -A && git commit -m "checkpoint: overnight-start $(date +%Y%m%d_%H%M%S)" || true

# Start in tmux if not already
if [ -z "$TMUX" ]; then
    tmux new-session -d -s overnight -c "$PROJECT_DIR"
    tmux send-keys -t overnight "$0 '$TASK'" C-m
    echo "Started in tmux session 'overnight'"
    exit 0
fi

# Launch containerized Claude Code
docker-compose up -d

docker exec -it claude-overnight claude \
    --dangerously-skip-permissions \
    -p "$TASK" 2>&1 | tee -a "$LOG_FILE"

# Post-session checkpoint
git add -A && git commit -m "checkpoint: overnight-end $(date +%Y%m%d_%H%M%S)" || true

# Notify completion
python3 scripts/notify.py complete "Overnight session" 0 true
```

Scheduling with cron

Add to crontab (`crontab -e`):

```bash
```

```
# Start overnight session at 10 PM
0 22 * * * /home/user/projects/trading-bot/scripts/overnight-session.sh "Implement RSI divergence indicator"

# Stop at 6 AM if still running
0 6 * * * docker stop claude-overnight 2>/dev/null || true
```

Conclusion

Autonomous overnight Claude Code operation requires four integrated systems working together: **persistent WSL2 sessions via tmux**, **Docker sandboxing for safe permissions bypass**, **hooks for automation and validation**, and **CLAUDE.md for project-specific intelligence**. The critical insight is that safety and autonomy aren't opposed—proper containerization enables more aggressive automation than would otherwise be prudent.

For QuantConnect development specifically, integrate LEAN backtesting into your validation hooks so strategies are tested automatically before commits. Use checkpoint-based git workflows to enable recovery from any failed changes. And implement proactive context management through document-and-clear patterns rather than relying on auto-compact.

Start with the minimal configuration: a Docker container with network isolation, a well-structured CLAUDE.md, and basic logging. Add monitoring, notifications, and recovery tooling as you build confidence in the system. The community tools like CCAutoRenew and claude-auto-resume can help bridge gaps in session management while the platform continues to evolve.